ADVANCES IN COMPUTER GAMES

Many Games, Many Challenges

Proceedings of the 10th Advances in Computer Games Conference (ACG-10)

Edited by
H. Jaap van den Herik
Hiroyuki Iida
Ernst A. Heinz
ADVANCES IN COMPUTER GAMES
IFIP – The International Federation for Information Processing

IFIP was founded in 1960 under the auspices of UNESCO, following the First World Computer Congress held in Paris the previous year. An umbrella organization for societies working in information processing, IFIP's aim is two-fold: to support information processing within its member countries and to encourage technology transfer to developing nations. As its mission statement clearly states,

*IFIP's mission is to be the leading, truly international, apolitical organization which encourages and assists in the development, exploitation and application of information technology for the benefit of all people.*

IFIP is a non-profitmaking organization, run almost solely by 2500 volunteers. It operates through a number of technical committees, which organize events and publications. IFIP's events range from an international congress to local seminars, but the most important are:

- The IFIP World Computer Congress, held every second year;
- Open conferences;
- Working conferences.

The flagship event is the IFIP World Computer Congress, at which both invited and contributed papers are presented. Contributed papers are rigorously refereed and the rejection rate is high.

As with the Congress, participation in the open conferences is open to all and papers may be invited or submitted. Again, submitted papers are stringently refereed.

The working conferences are structured differently. They are usually run by a working group and attendance is small and by invitation only. Their purpose is to create an atmosphere conducive to innovation and development. Refereeing is less rigorous and papers are subjected to extensive group discussion.

Publications arising from IFIP events vary. The papers presented at the IFIP World Computer Congress and at open conferences are published as conference proceedings, while the results of the working conferences are often published as collections of selected and edited papers.

Any national society whose primary activity is in information may apply to become a full member of IFIP, although full membership is restricted to one society per country. Full members are entitled to vote at the annual General Assembly, National societies preferring a less committed involvement may apply for associate or corresponding membership. Associate members enjoy the same benefits as full members, but without voting rights. Corresponding members are not represented in IFIP bodies. Affiliated membership is open to non-national societies, and individual and honorary membership schemes are also offered.
ADVANCES IN COMPUTER GAMES

Many Games, Many Challenges

Proceedings of the ICGA / IFIP SG16
10th Advances in Computer Games Conference (ACG 10)
November 24–27, 2003, Graz, Styria, Austria

Edited by

H. JAAP VAN DEN HERIK
Universiteit Maastricht, IKAT
The Netherlands

HIROYUKI IIDA
Shizuoka University, Hamamatsu
Japan

ERNST A. HEINZ
Frankfurt a.M.
Germany

SPRINGER SCIENCE+BUSINESS MEDIA, LLC
## Contents

Foreword .......................................................... vii
Preface .................................................................. ix

Evaluation Function Tuning via Ordinal Correlation ................. 1
*D. Gomboc, T.A. Marsland, M. Buro*

First Experimental Results of ProbCut Applied to Chess ............. 19
*A.X. Jiang, M. Buro*

Search versus Knowledge: An Empirical Study of Minimax on KRK 33
*A. Sadikov, I. Bratko, I. Kononenko*

Static Recognition of Potential Wins in KNNKB and KNNKN ......... 45
*E.A. Heinz*

Model Endgame Analysis .................................................. 65
*G.M°C. Haworth, R.B. Andrist*

Chess Endgames: Data and Strategy ........................................ 81
*J.A. Tamplin, G.M°C. Haworth*

Evaluation in Go by a Neural Network using Soft Segmentation .... 97
*M. Enzenberger*

When One Eye is Sufficient: A Static Classification .................... 109
*R. Vilà, T. Cazenave*

DF-PN in Go: An Application to the One-Eye Problem ................ 125
*A. Kishimoto, M. Müller*

Learning to Score Final Positions in the Game of Go ................. 143
*E.C.D. van der Werf, H.J. van den Herik, J.W.H.M. Uiterwijk*

Monte-Carlo Go Developments ............................................. 159
*B. Bouzy, B. Helmstetter*
Static Analysis by Incremental Computation in Go Programming ............. 175
K. Nakamura

Building the Checkers 10-piece Endgame Databases ......................... 193
J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, S. Sutphen

The 7-piece Perfect Play Lookup Database for the Game of Checkers ....... 211
E. Trice, G. Dodgen

Search and Knowledge in Lines of Action .................................... 231
D. Billings, Y. Björnsson

An Evaluation Function for Lines of Action .................................. 249
M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk

Solving $7 \times 7$ Hex: Virtual Connections and Game-State Reduction ....... 261
R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, J. van Rijswijck

Automated Identification of Patterns in Evaluation Functions ................. 279
T. Kaneko, K. Yamaguchi, S. Kawai

An Evaluation Function for the Game of Amazons ............................ 299
J. Lieberum

Opponent-Model Search in Bao: Conditions for a Successful Application ...... 309

Computer Programming of Kriegspiel Endings: The Case of KR versus K ..... 325
A. Bolognesi, P. Ciancarini

Searching with Analysis of Dependencies in a Solitaire Card Game .......... 343
B. Helmstetter, T. Cazenave

Solving the Oshi-Zumo Game ................................................. 361
M. Buro

New Games Related to Old and New Sequences ............................... 367
A.S. Fraenkel

Author Index .............................................................................. 383
Foreword

I feel privileged that the 10th Advances in Computer Games Conference (ACG 10) takes place in Graz, Styria, Austria. It is the first time that Austria acts as host country for this major event. The series of conferences started in Edinburgh, Scotland in 1975 and was then held four times in England, three times in The Netherlands, and once in Germany. The ACG-10 conference in Graz is special in that it is organised together with the 11th World Computer-Chess Championship (WCCC), the 8th Computer Olympiad (CO), and the European Union Youth Chess Championship.

The 11th WCCC and ACG 10 take place in the Dom im Berg (Dome in the Mountain), a high-tech space with multimedia equipment, located in the Schlossberg, in the centre of the city. The help of many sponsors (large and small) is gratefully acknowledged. They will make the organisation of this conference a success. In particular, I would like to thank the European Union for designating Graz as the Cultural Capital of Europe 2003. There are 24 accepted contributions by participants from all over the world: Europe, Japan, USA, and Canada. The specific research results of the ACG 10 are expected to find their way to general applications. The results are described in the pages that follow. The international stature together with the technical importance of this conference reaffirms the mandate of the International Computer Games Association (ICGA) to represent the computer-games community. This is important when negotiating with FIDE or other representative bodies of game competitions on the organisation of a match against their domain-specific human World Champion. Moreover, the ICGA is the right organisation to represent the same community to the European Union to have the next series of events (WCCC, CO, ACG) organised in the framework of the Cultural Capital of Europe. I would hope that Graz is the start of such a trend. I am convinced that our city will do its utmost to let the participants feel at ease when they, for a moment, are not in the brain-teasing theories and experiments of their brainchild. In summary, I wish you a good time in Graz.

Kurt Jungwirth
Organising Chair of the ACG 10 in Graz

September 2003
Preface

This book is the tenth in a well-established series originally describing the progress of computer-chess research. The book contains the papers of the 10\textsuperscript{th} international conference \textit{Advances in Computer Games} (ACG), to be hosted by the city of Graz (Styria, Austria), the Cultural Capital of Europe 2003. The conference will take place from November 24 to 27, 2003 during the 11\textsuperscript{th} World Computer-Chess Championship (WCCC) and the 8\textsuperscript{th} Computer Olympiad, which will be held simultaneously in Graz. The combination of the three events is expected to be a great success since it offers: science, competition, and top sport (in the domain of computer chess). It is the first time that the three events coincide. For Graz it is very fortunate that the ICGA (International Computer Games Association) decided in its Triennial Meeting in Maastricht 2002 to have the WCCC annually instead of triennially.

In the last decade of the previous century the focus of much academic research shifted from chess to other intelligent games. Perhaps, the two matches Kasparov played with DEEP BLUE were instrumental for this shift. Whatever the reason, it is obvious that the oriental game of Go currently plays a considerable part in intelligent games research. The tendency is clearly visible in the 10\textsuperscript{th} ACG conference, where chess and Go are represented by an equal amount of contributions. For historical reasons we start with chess, still turning out to be an inexhaustible testing ground for new ideas.

The book contains 24 contributions by a variety of authors from all over the world. We have sequenced the contributions according to the type of game. As stated above we start with the research domains of chess (6 papers) and Go (6 papers). It is followed by those of checkers (2 papers) and Lines of Action (2 papers). Finally, we are happy to show the broadness of the 10\textsuperscript{th} ACG conference by publishing another eight contributions on different games each. They are: Hex, Othello, Amazons, Bao, Kriegspiel, Gaps, Oshi-Zumo, and New Wythoff games. We hope that our readers will enjoy reading the efforts of the researchers, who made this development possible. Below we give a brief account of all contributions.
Chess

Chess is a game that has set the AI research scene for almost fifty years. The game dominated the games developments to a large extent. Since chess can hardly be characterized by a limited list of research topics, we are happy and surprised that the topics are completely different. The six contributions deal with (1) evaluation functions, (2) pruning of the search, (3) search and knowledge, (4) pattern recognition, (5) modelling, and (6) strategies.

In Evaluation Function Tuning via Ordinal Correlation, Dave Gomboc, Tony Marsland, and Michael Buro discuss the heart of any chess program: the evaluation function. They arrive at a metric for assessing the quality of a static evaluation function. Their application of ordinal correlation is fundamentally different from prior evaluation-function tuning techniques.

In First Experimental Results of ProbCut Applied to Chess, Albert Xin Jiang and Michael Buro show that Multi-ProbCut is a technique not only successful in Othello and Shogi, but also in chess. The contribution discusses details of the implementation in the chess engine CRAFTY. The recorded results state that the new version wins over the original one with a 59 percent score in their test setup.

In Search versus Knowledge: An Empirical Study of Minimax on KRK, Alexander Sadikov, Ivan Bratko, and Igor Kononenko return to the old research topic of intricacies of the precise working of the minimax algorithm. Their empirical experiment throws a new light on this topic.

In Static Recognition of Potential Wins in KNNKB and KNNKN, Ernst Heinz investigates the possibilities of how to recognize surprisingly tricky mate themes in the endgames named. He analyses the mate themes and derives rules from them which allow for a static recognition. He shows that such positions occur more frequently than generally assumed.

In Model Endgame Analysis, Guy Haworth and Rafael Andrist introduce a reference model of fallible endgame play. The results are compared with a Markov model of the endgame in question and are found to be in close agreement with those of the Markov model.

In Chess Endgames: Data and Strategy, John Tamplin and Guy Haworth compare Nalimov's endgame tablebases with newly created tables in which alternative metrics have been applied. The research is on measuring the differences in strategy.

Go

The six contributions on the game of Go relate to the following general topics: (1) evaluation, (2) eyes, (3) search, (4) learning, (5) Monte-Carlo Go, and (6) static analysis.
In *Evaluation in Go by a Neural Network using Soft Segmentation*, Markus Enzenberger presents a network architecture that is applied to position evaluation. It is trained using self-play and temporal-difference learning combined with a rich two-dimensional reinforcement signal. One of the methods is able to play at a level comparable to a 13-kyu Go program.

In *When One Eye is Sufficient: A Static Classification*, Ricard Vila and Tristan Cazenave propose a new classification for eye shapes. The method is said to replace a possibly deep tree by a fast, reliable and static evaluation.

In *DF-PN in Go: An Application to the One-Eye Problem*, Akihiro Kishimoto and Martin Müller modify the depth-first proof-number search algorithm and apply it to the game of Go. Subsequently, they develop a solver for one-eye problems.

In *Learning to Score Final Positions in the Game of Go*, Erik van der Werf, Jaap van den Herik, and Jos Uiterwijk present a learning system that scores 98.9 per cent of the submitted positions correctly. Such a reliable scoring method opens the large source of Go knowledge and thus paves the way for a successful application in machine learning in Go.

In *Monte-Carlo Go Developments*, Bruno Bouzy and Bernard Helmstetter report on the development of two Go programs OLGA and OLEG. The authors perform experiments to test their ideas on progressive pruning, temperature, and depth-two tree search within the Monte-Carlo framework. They conclude that such approaches are worth to be considered in future research.

In *Static Analysis by Incremental Computation in Go Programming*, Katsuhiko Nakamura describes two types of analysis and pattern recognition. One is based on the determination of groups almost settled, the other on an estimation of groups of stones and territories by analysing the influence of stones using the "electric charge" model.

**Checkers**

Both contributions on the game of checkers focus on endgame databases.

In *Building the Checkers 10-piece Endgame Databases*, Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Robert Lake, Paul Lu, and Steve Sutphen report on their results of building large endgame databases. They describe actions as compression, data organisation, and real-time decompression. It is amazing to see that powerful techniques and machine power in itself are just not sufficient to crack the game.

In *The 7-piece Perfect Play Lookup Database for the Game of Checkers*, Edward Trice and Gilbert Dodgen examine the benefits and detriments associated with computing three different types of checkers endgame databases. They show major improvements to some previously published play.
Lines of Action

Two contributions concentrate on Lines of Action (LoA).

In *Search and Knowledge in Lines of Action*, Darse Billings and Yngvi Björnsson provide accurate descriptions on the design and development of the programs YL and MONA. YL emphasizes fast and efficient search, whereas MONA focuses on a sophisticated but relatively slow evaluation. It is an ideal relation for the investigation of the trade-off between search and knowledge. The results concur with well-known results from the chess world: (1) diminishing returns with additional search depth, and (2) the knowledge level of a program has a significant impact on the results.

In *An Evaluation Function for Lines of Action*, Mark Winands, Jaap van den Herik, and Jos Uiterwijk, extensively describe the evaluation function that brought MIA IV (Maastricht In Action) its successes. The important elements are: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player-to-move. In the experiments, the evaluation function performs better at deeper searches showing the relevance of the components.

Hex

*Solving 7x7 Hex: Virtual Connections and Game-State Reduction* is a team effort by Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. They develop an algorithm that determines the outcome of an arbitrary Hex game-state. The algorithm is based on the concept of a proof tree.

Othello

In *Automated Identification of Patterns in Evaluation Functions*, Tomoyuki Kaneko, Kazunori Yamaguchi, and Satoru Kawai propose a method that generates accurate evaluation functions using patterns, without expert players’ knowledge. The approach consists of three steps (generation of logical features, extracting of patterns, and selection of patterns) and is applied to the game of Othello. The authors report the successes of their method and claim that the accuracy is comparable to that of specialized Othello programs.

Amazons

In *An Evaluation Function for the Game of Amazons*, Jens Lieberum reveals the secrets of his program that won the Computer Olympiad in Maastricht 2002. The secret is the evaluation function. More on this topic can be found in the work itself.
Bao
In Opponent-Model Search in Bao: Conditions for a Successful Application, Jeroen Donkers, Jaap van den Herik, and Jos Uiterwijk investigate the role of prediction and estimation. The rules of Bao are described and five evaluation functions are tested in tournaments. The domain of research is variable with respect to all kinds of versions of opponent modelling. The final result is that opponent-model search can be applied successfully, provided that the conditions are met.

Kriegspiel
In Computer Programming of Kriegspiel Endings: The Case of KR versus K, Andrea Bolognesi and Paolo Ciancarini describe the rationale and the design of a Kriegspiel program that plays the ending King and Rook versus King adequately.

Gaps
In Searching with Analysis of Dependencies in a Solitaire Card Game, Bernard Helmstetter and Tristan Cazenave present a new method of playing the card game Gaps. The method is an improvement of depth-first search by grouping several positions in a block and searching only on the boundaries of the blocks.

Oshi Zumo
In Solving the Oshi-Zumo Game, Michael Buro completes a previous analysis by Kotani. Buro’s Nash-optimal mixed strategies are non-trivial, but can be computed quickly. A discussion on 'how good is optimal?' concludes the article.

New Wythoff Games
In New Games Related to Old and New Sequences, Aviezri Fraenkel defines an infinite class of 2-pile subtraction games, where the amount that can be subtracted from both piles simultaneously is a function $f$ of the size of the piles. Wythoff’s game is a special case. The author introduces new sequences. The main result is a theorem giving necessary and sufficient conditions on $f$ so that the sequences are 2nd player winning positions.

Acknowledgements
This book would not have been produced without the help of many persons. In particular we would like to mention the authors and the referees. Moreover, the organisers of the festivities in Graz have contributed also quite substantially by bringing the researchers together. A special word of thanks goes to the organisation committee of the ACG 10, consisting of Kurt
Jungwirth (chair), Johanna Hellemens, and Martine Tiessen. On top of these thanks, the Editors happily recognise the generous sponsorship by the European Union, financially supporting the conference. With much pleasure we mention that the ACG 10 takes places under the aegis of the ICGA and the IFIP. In particular, the Specialist Group SG-16 of the International Federation of Information Processing is involved.


Finally, we would like to express our sincere gratitude to Jeroen Donkers, Hazel den Hoed, Martine Tiessen, and Erik van der Werf for their assistance during the editing process, especially in the final stage of preparing this collection of contributions for publication.

Jaap van den Herik
Hiroyuki Iida
Ernst Heinz

Maastricht, September 2003
EVALUATION FUNCTION TUNING VIA ORDINAL CORRELATION

D. Gomboc, T. A. Marsland, M. Buro
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

Abstract  Heuristic search effectiveness depends directly upon the quality of heuristic evaluations of states in the search space. We show why ordinal correlation is relevant to heuristic search, present a metric for assessing the quality of a static evaluation function, and apply it to learn feature weights for a computer chess program.

Keywords: ordinal correlation, Kendall's τ (tau), static evaluation function, heuristic search, computer chess

1. Introduction

Inspiration for this research came while reflecting on how evaluation functions for today's computer chess programs are usually developed. Typically, evaluation functions are refined over many years, based upon careful observation of their performance. During this time, engine authors will tweak feature weights repeatedly by hand in search of proper balance between terms. This ad hoc process is used because the principal way to measure the utility of changes to a program is to play many games against other programs and interpret the results. The process of evaluation function development would be considerably assisted by the presence of a metric that could reliably indicate a tuning improvement. But what would such a metric be like?

The critical operation of minimax game-tree searches (Shannon, 1950) and all its derivatives (Marsland, 1983; Plaat, 1996) is the asking of a single question: is position B better than position A? Note that it is not “How much better?”, but simply “Is it better?”. In minimax, instead of propagating values one could propagate the positions instead, and, as humans do, choose between them directly without using values as an intermediary.
Consequently, we need only pairwise comparisons that tell us whether B is preferable to A. Plausibly, then, the metric we seek will assess how well an evaluation function orders positions in relation to each other, without placing importance on the relative differences in the values of the assessed positions— that is, it will be ordinal in nature.

While at shallow depths some resemblance between positions compared by a minimax-based search will be evident, this does not hold true at the search depths typically reached today. The positions that are being compared are frequently completely different in character, suggesting that our mystery metric ought to compare pairs of positions not merely from local pockets of the search space but globally.

Consideration was also given to harnessing the great deal of recorded experience of human chess for developing a static evaluation function. Researchers have tried to make their machines play designated moves from test positions, but we focus on judgments about the relative worth of positions, reasoning that if these are correct then strong moves will emerge as a consequence. But how does one compute a correlation between the (ordinal) human assessment symbols, given in Table 1, with machine assessments? A literature review identified that a statistical measure known as Kendall's $\tau$ might be exactly what is needed.

After a brief overview of prior work on the automated tuning of static evaluation functions, we describe Kendall's $\tau$, and our novel algorithm to implement it efficiently. We then discuss the materials used for our experiments, followed by details of our software implementation. Experimental results are provided in Section 6. After drawing some conclusions, we suggest further investigations to the interested researcher.

---

**Table 1.** Symbols for chess position assessment.

<table>
<thead>
<tr>
<th>symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rightarrow$</td>
<td>white is winning</td>
</tr>
<tr>
<td>$\pm$</td>
<td>white has a clear advantage</td>
</tr>
<tr>
<td>$\pm$</td>
<td>white has an edge</td>
</tr>
<tr>
<td>$=$</td>
<td>the position is equal</td>
</tr>
<tr>
<td>$\mp$</td>
<td>black has a clear advantage</td>
</tr>
<tr>
<td>$\pm$</td>
<td>black has an edge</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>black is winning</td>
</tr>
</tbody>
</table>

2. Prior Work

The precursor of modern machine learning in games is the work done by Samuel (1959, 1967). By fixing the value for a checker advantage, while letting other weights float, he iteratively tuned the weights of evaluation

---

1 Two other assessment symbols, $\infty$ (the position is unclear) and $\approx$ (a player has positional compensation for a material deficit) are also frequently encountered. Unfortunately, the usage of these two symbols is not consistent throughout chess literature. Accordingly, we ignore positions labeled with these assessments.
function features so that the assessments of predecessor positions became more similar to the assessments of successor positions.

Hartmann (1989) developed the “Dap Tap” to determine the relative influence of various evaluation feature categories, or notions, on the outcome of chess games. Using 62,965 positions from grandmaster tournament and match games, he found that “the most important notions yield a clear difference between winners and losers of the games”. Unsurprisingly, the notion of material was predominant; the combination of other notions contribute roughly the same proportion to the win as material did alone. He further concluded that the threshold for one side to possess a decisive advantage is 1.5 pawns.

The DEEP THOUGHT (later DEEP BLUE) team applied least squares fitting to the moves of the winners of 868 grandmaster games to tune their evaluation function parameters as early as 1987 (Nowatzyk, 2000). They found that tuning to maximize agreement between their program’s preferred choice of move and the grandmaster’s was “not really the same thing” as playing more strongly. Amongst other interesting observations, they discovered that conducting deeper searches while tuning led to superior weight vectors being reached.

Tesauro (1995) initially configured a neural network to represent the backgammon state in an efficient manner, and trained it via temporal difference learning (Sutton, 1988). After 300,000 self-play games, the program reached strong amateur level. Subsequent versions also contained hidden units representing specialized backgammon knowledge and used minimax search. TD-GAMMON is now a world-class backgammon player.

Beal and Smith (1997) applied temporal difference learning to determine piece values for a chess program that included material, but not positional, terms. Program versions using weights resulting from five randomized self-play learning trials each won a match versus a sixth program version that used the conventional weights given in most introductory chess texts. They have since extended their reach to include piece-square tables for chess (Beal and Smith, 1999a) and piece values for Shogi (Beal and Smith, 1999b).

Baxter, Tridgell, and Weaver (1998) applied temporal difference learning to the leaves of the principal variations returned by alpha-beta searches to learn feature weights for their program KNIGHTCAP. Through online play against humans, KNIGHTCAP’s skill level improved from beginner to strong master. The authors credit this to: the guidance given to the learner by the varying strength of its pool of opponents, which improved as it did; the exploration of the state space forced by stronger opponents who took advantage of KNIGHTCAP’s mistakes; the initialization of material values to reasonable settings, locating KNIGHTCAP’s weight vector “close in parameter space to many far superior parameter settings”.

Buro (1995) estimated feature weights by performing logistic regression on win/loss/draw-classified Othello positions. The underlying log-linear model is well suited for constructing evaluation functions for approximating winning probabilities. In that application, it was also shown that the evaluation function based on logistic regression can perform better than those based on linear and quadratic discriminant functions. Later, Buro (1999) presented a much superior approach, using linear regression and positions labeled with the final disc differential to optimize the weights of thousands of binary pattern features.

Kendall and Whitwell (2001) evolved intermediate-strength players from a population of poor players by applying crossover and mutation operators to generate new weight vectors, while discarding vectors that performed poorly.

3. Kendall’s Tau

Concordance, or agreement, occurs where items are ranked in the same order. Kendall’s $\tau$ is all about the similarities and differences in the ordering of ordered pairs. Consider two pairs, $(x_i, y_i)$ and $(x_k, y_k)$. Compare both the $x$ values and the $y$ values. Table 2 defines the relationship between the pairs.

<table>
<thead>
<tr>
<th>Relationship between $x_i$ and $x_k$</th>
<th>Relationship between $y_i$ and $y_k$</th>
<th>Relationship between $(x_i, y_i)$ and $(x_k, y_k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i &lt; x_k$</td>
<td>$y_i &lt; y_k$</td>
<td>Concordant</td>
</tr>
<tr>
<td>$x_i &lt; x_k$</td>
<td>$y_i &gt; y_k$</td>
<td>Discordant</td>
</tr>
<tr>
<td>$x_i &gt; x_k$</td>
<td>$y_i &lt; y_k$</td>
<td>Discordant</td>
</tr>
<tr>
<td>$x_i &gt; x_k$</td>
<td>$y_i &gt; y_k$</td>
<td>Concordant</td>
</tr>
<tr>
<td>$x_i = x_k$</td>
<td>$y_i \neq y_k$</td>
<td>extra $y$ pair</td>
</tr>
<tr>
<td>$x_i \neq x_k$</td>
<td>$y_i = y_k$</td>
<td>extra $x$ pair</td>
</tr>
<tr>
<td>$x_i = x_k$</td>
<td>$y_i = y_k$</td>
<td>duplicate pair</td>
</tr>
</tbody>
</table>

Table 2. Relationships between ordered pairs.

Table 3 contains a grid representing ordered pairs of machine and human evaluations. The value in each cell indicates the number of corresponding pairs; blank cells indicate that no such pairs are in the data set. Sample machine and human assessments are on the $x$ and $y$ axes, respectively.

To compute $\tau$ for a collection of ordered pairs, each ordered pair is compared against all other pairs. The total number of concordant pairs is designated $S^+$ ("S-positive"). Similarly, the total number of discordant pairs is designated $S^-$ ("S-negative").

Consider the table cell $(0.0, \ =)$. There are six entries, containing seven data points, located strictly below and to its left; these are concordant pairs and so contribute to $S^+$. The two discordant pairs, strictly below and to its right, contribute to $S^-$. We do not consider any cells from above the cell of
interest. If we did so, we would end up comparing each pair of ordered pairs twice instead of once. Finally, the 2 contained in the cell indicates that there are two (0,0, =) data points; hence the examination of this cell has produced $7 \times 2 = 14$ concordant pairs, and $2 \times 2 = 4$ discordant pairs.

$$
\begin{array}{cccccccccc}
& -1.6 & -1.1 & -0.7 & -0.6 & -0.3 & -0.1 & 0.0 & 0.1 & 0.2 & 0.3 & 0.5 & 1.3 \\
< & 1 & 1 & 1 & 1 \\
\leq & 1 & 1 & 1 & 2 & 1 & 2 \\
= & 1 & 1 & 1 & 1 \\
\geq & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\
\rightarrow & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
$$

Table 3. (machine, human) assessments, $n = 25$.

$\tau$ is given by:

$$
\tau = \frac{S^+ - S^-}{n(n-1)/2}
$$

The denominator equals the number of unique possible comparisons between any two ordered pairs from a collection of $n$ ordered pairs.

For the data in Table 3, $S^+$ is 162, $S^-$ is 83, and $n$, the number of ordered pairs, is 25. $\tau$ equals 0.2633; we might also say that the concordance of the data is 0.2633. Possible concordance values range from +1, representing complete agreement in ordering, to -1, representing complete disagreement in ordering. Whenever there are extra or duplicate pairs, the values of +1 and -1 are not achievable.

Cliff (1996) provides a more detailed exposition of Kendall's $\tau$, discussing variations thereof that optionally disregard extra and duplicate pairs. Cliff labels what we call $\tau$ as $\tau_a$, and uses it most often, noting that it has the simplest interpretation of the lot.

A straightforward implementation would perform the process illustrated above for each cell of the table. Our novel, algorithmically superior implementation allocates additional memory space, and in successive single passes through the data, applies dynamic programming to compute tables containing the number of data points that are:

- either on the same row as or below the current cell;
- either on the same column or to the right of the current cell;
- either on the same column or to the left of the current cell;
- strictly below and to the right of the current cell;
- strictly below and to the left of the current cell.

Then, in a final pass, $S^+$ and $S^-$ are computed by multiplying the number of data points in the current cell by the data in the final two tables listed. It is
also possible to use more passes, but less memory, by performing the sweeps to the left and to the right serially instead of in parallel.

There is a better-known ordinal metric in common use: Spearman’s ρ, also known as Spearman correlation. In our application, the number of distinct human assessments is constant. Therefore, after initial data processing has identified the unique machine assessments for memory allocation and indexing purposes, τ is computed in time linear in the number of unique machine assessments, which is not possible for ρ. Prototype implementations confirmed that τ was significantly quicker to compute for large data sets.

Not only does τ more directly measure what interests us ("for all pairs of positions (A, B), is position B better than position A?"), it is also more efficient to compute than plausible alternatives. Therefore, we concentrate on τ in this paper.

4. Chess-Related Components

Many chess programs, or chess engines, exist. Some are commercially available; most are hobbyist. For our work, we selected CRAFTY, by Robert Hyatt (1996) of the University of Alabama. CRAFTY is the best chess engine choice for our work for several reasons: the source was readily available to us, facilitating experimentation; it is the strongest such open-source engine today; previous research has already been performed using CRAFTY. We worked with version 19.1 of the program.

4.1 Training Data

To assess the correlation of τ with improved play, we used 649,698 positions from Chess Informant 1 through 85 (Sahovski, 1966). These volumes cover the important chess games played between January 1966 and September 2002. This data set was selected because it contains a variety of assessed positions from modern grandmaster play, the assessments are made by qualified individuals, it is accessible in a non-proprietary electronic form, and chess players around the world are familiar with it.

We used a 32,768-position subset for the preliminary feature weight tuning experiments reported here.

4.2 Test Suites

English chess grandmaster John Nunn (1999) developed the Nunn and Nunn II test suites of 10 and 20 positions, respectively. They serve as starting positions for matches between computer chess programs, where the
experimenter is interested in the engine’s playing skill independent of the quality of its opening book. Nunn selected positions that are approximately balanced, commonly occur in human games, and exhibit variety of play. We refer to these collectively as the “Nunn 30”.

Don Dailey, known for his work on STARSOCRATES and CILKCHESS, prepared a file of two hundred commonly reached positions, all of which are ten ply from the initial position. We refer to these collectively as the “Dailey 200”.

5. Software Implementation

Here we detail some specifics of our implementation. We discuss both alterations made to CRAFTY and new software written as a platform for our experiments.

5.1 Use of Floating-Point Computation

We modified CRAFTY so that variables holding machine assessments are declared to be of an aliased type rather than directly as integers. This allows us to choose whether to use floating-point or integer arithmetic via a compilation switch. The use of floating-point computation provides a learning environment where small changes in values can be rewarded. With these modifications, CRAFTY is slower, but only by a factor of two to three on a typical personal computer. The experiments were performed with this modified version; however, all test matches were performed with the original, integer-based evaluation implementation. Further details can be found in Section 6.

It might strike the reader as odd that we chose to alter CRAFTY in this manner rather than scaling up all the evaluation function weights. There are significant practical disadvantages to that approach. How would we know that everything had been scaled? It would be easy to miss some value that needed to be changed. How would we identify overflow issues? It might be necessary to switch to a larger integer type. How would we know that we had scaled up the values far enough? It would be frustrating to have to repeat the procedure.

By contrast, the choice of converting to floating-point is safer. Precision and overflow are no longer concerns. Also, by setting the typedef to be a non-arithmetic type we can cause the compiler to emit errors wherever type mismatches exist. Thus, we can be more confident that our experiments rest upon a sound foundation.
5.2 Hill Climbing

We implemented an iteration-based learner, and a hill-climbing algorithm. Other iteration-based algorithms may be substituted for the hill-climbing code if desired. Because we are not working with an analytic function, we measure the gradient empirically.

We multiply $V_{\text{current}}$, the current weight of a feature being tuned, by a number fractionally greater than one\(^1\) to get $V_{\text{high}}$, except when $V_{\text{current}}$ is near zero, in which case a minimum distance between $V_{\text{current}}$ and $V_{\text{high}}$ is enforced. $V_{\text{low}}$ is then set to be equidistant from $V_{\text{current}}$, but in the other direction, so that $V_{\text{current}}$ is bracketed between $V_{\text{low}}$ and $V_{\text{high}}$. Two test weight vectors are generated: one using $V_{\text{high}}$, the other using $V_{\text{low}}$. All other weights for these test vectors remain the same as in the base vector. This procedure is performed for each weight that is being tuned. For example, when 11 parameters are being learned, $1 + 11 \times 2 = 23$ vectors are examined per iteration: the base vector, and 22 test vectors.

The three computed concordances related to a weight being tuned ($\tau_{\text{current}}$, $\tau_{\text{low}}$, and $\tau_{\text{high}}$) are then compared. If all three are roughly equal, no change is made: we select $V_{\text{current}}$. If $\tau_{\text{current}}$ is lower than both $\tau_{\text{low}}$ and $\tau_{\text{high}}$, we choose the $V$ corresponding to the highest $\tau$. If they are in either increasing or decreasing order, we use the slope of test points ($V_{\text{low}}, \tau_{\text{low}}$) and ($V_{\text{high}}, \tau_{\text{high}}$) to interpolate a new point. However, to avoid occasional large swings in parameter settings, we bound the maximum change from $V_{\text{current}}$. The final case occurs when $\tau_{\text{current}}$ is higher than both $\tau_{\text{low}}$ and $\tau_{\text{high}}$. In this case, we apply inverse parabolic interpolation to select the apex of the parabola formed by the three points, in the hope that this will lead us to the highest $\tau$ in the region.

Once this procedure has been performed for all of the weights being learned, it is possible to postprocess the weight changes, for instance to normalize them. However, at present we have not found this to be necessary. The chosen values now become the new base vector for the next iteration.

5.3 Automation

A substantial amount of code was written to automate the communication of work and results between multiple, distributed instantiations of CRAFTY and the PostgreSQL database. We implemented placeholder scheduling (Pinchak, 2002) so that learning could occur more rapidly, and without human intervention.

\(^1\) The tuning experiments reported in this paper used 1.01.
5.4 Search Effort Quantum

Traditionally, researchers have used search depth to quantify search effort. For our learning algorithm, doing so would not be appropriate: the amount of effort required to search to a fixed depth varies wildly between positions, and we will be comparing the assessments of these positions. However, because we did not have the dedicated use of computational resources, we could not use search time either. While it is known that chess engines tend to search more nodes per second in the endgame than the middlegame, this difference is insignificant for our short searches because it is dwarfed by the overhead of preparing the engine to search an arbitrary position. Therefore, we chose to quantify search effort by the number of nodes visited.

We instructed CRAFTY to search 16,384 nodes to assess a position. Earlier experiments that directly called the static evaluation or quiescence search routines to form assessments were not successful. When searching 1,024 nodes per position, we had mixed results. Like the DEEP THOUGHT team (Nowatzyk, 2000), we found that larger searches improve the quality of learning. The downside is, of course, the additional processor time required by the learning process.

There are positions in our data set from which CRAFTY does not complete a 1-ply search within 16,384 nodes, because its quiescence search explores many sequences of captures. When this occurs, no evaluation score is available to use. Instead of using either zero or the statically computed evaluation (which is not designed to operate without a quiescence search), we chose to throw away the data point for that particular computation of $\tau$, reducing the position count ($n$). However, the value of $\tau$ for similar data of different population sizes is not necessarily constant. As feature weights are changed, the shape of the search tree for positions may also change. This can cause CRAFTY to not finish a 1-ply search for a position within the node limit where it was previously able to do so, or vice versa. When many transitions in the same direction occur simultaneously, noticeable irregularities are introduced into the learning process. Ignoring the node count limitation until the first ply of search has been completed may be a better strategy.

5.5 Performance

Early experiments were performed using idle time on various machines in our department. Lately, we have had (non-exclusive) access to clusters of personal computer workstations, which is helpful because the task of computing $\tau$ for distinct weight vectors within an iteration is trivially parallel. Examining 32,768 positions and computing $\tau$ takes about two
minutes per weight vector. The cost of computing \( \tau \) is negligible in comparison, so in the best case, when there are enough nodes available for the concordances of all weight vectors of an iteration to be computed simultaneously, learning proceeds at the rate of 30 iterations per hour.

6. Experimental Results

We demonstrate that concordance between human judgments and machine assessments increases with increasing depth of machine search. This result, combined with knowing that play improves as search depth increases (Thompson, 1982), in turn justifies our attempt to use this concordance as a metric to tune selected feature weights of CRAFTY's static evaluation function.

6.1 Concordance as Machine Search Effort Increases

In Table 4 we computed \( \tau \) for depths 1 through 7 for \( n = 649,698 \) positions, performing work equivalent to 211 billion \((10^9)\) comparisons at each depth. \( S^+ \) and \( S^- \) are reported in billions. As search depth increases, the difference between \( S^+ \) and \( S^- \), and therefore \( \tau \), also increases. The sum of \( S^+ \) and \( S^- \) is not constant because at different depths different amounts of extra y-pairs and duplicate pairs are encountered.

<table>
<thead>
<tr>
<th>depth</th>
<th>( S^+ / 10^9 )</th>
<th>( S^- / 10^9 )</th>
<th>( \tau )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>110.374</td>
<td>65.298</td>
<td>0.2136</td>
</tr>
<tr>
<td>2</td>
<td>127.113</td>
<td>48.934</td>
<td>0.3705</td>
</tr>
<tr>
<td>3</td>
<td>131.384</td>
<td>45.002</td>
<td>0.4093</td>
</tr>
<tr>
<td>4</td>
<td>141.496</td>
<td>36.505</td>
<td>0.4975</td>
</tr>
<tr>
<td>5</td>
<td>144.168</td>
<td>34.726</td>
<td>0.5186</td>
</tr>
<tr>
<td>6</td>
<td>149.517</td>
<td>30.136</td>
<td>0.5656</td>
</tr>
<tr>
<td>7</td>
<td>150.977</td>
<td>29.566</td>
<td>0.5753</td>
</tr>
</tbody>
</table>

It is difficult to predict how close an agreement might be reached using deeper searches. Two effects come into play: diminishing returns from additional search, and diminishing accuracy of human assessments relative to ever more deeply searched machine assessments. Particularly interesting is the odd-even effect on the change in \( \tau \) as depth increases. It has long been known that searching to the next depth of an alpha-beta search requires relatively much more effort when that next depth is even than when it is odd (Marsland, 1983). Notably, \( \tau \) tends to increase more in precisely these cases.

Similar experiments performed using increasing node counts, and increasing wall clock time (on a dedicated machine) with a different, smaller data set also gave increasing concordance, but, as expected, did not exhibit the staggered rise of the increasing depth searches. In sum, these experiments lend credibility to our belief that \( \tau \) is a direct measure of decision quality.
6.2 Tuning of CRAFTY’s Feature Weights

CRAFTY uses centipawns (hundredths of a pawn) as its evaluation function resolution, so experiments were performed by playing CRAFTY as distributed versus CRAFTY with the learned weights rounded to the nearest centipawn. Each program played each position both as White and as Black. The feature weights we tuned are given along with their default values in Table 5.

<table>
<thead>
<tr>
<th>feature</th>
<th>default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>king safety scaling factor</td>
<td>100</td>
</tr>
<tr>
<td>king safety asymmetry scaling factor</td>
<td>-40</td>
</tr>
<tr>
<td>king safety tropism scaling factor</td>
<td>100</td>
</tr>
<tr>
<td>blocked pawn scaling factor</td>
<td>100</td>
</tr>
<tr>
<td>passed pawn scaling factor</td>
<td>100</td>
</tr>
<tr>
<td>pawn structure scaling factor</td>
<td>100</td>
</tr>
<tr>
<td>bishop</td>
<td>300</td>
</tr>
<tr>
<td>knight</td>
<td>300</td>
</tr>
<tr>
<td>rook on the seventh rank</td>
<td>30</td>
</tr>
<tr>
<td>rook on an open file</td>
<td>24</td>
</tr>
<tr>
<td>rook behind a passed pawn</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5. Tuned features, with CRAFTY’s default values.

The scaling factors were chosen because they act as control knobs for many subterms. Bishop and knight were included because they participate in the most common piece imbalances. Trading a bishop for a knight is common, so it is important to include both to show that one is not learning to be of a certain weight chiefly because of the weight of the other. We also included three of the most important positional terms involving rooks. Material values for the rook and queen are not included because trials showed that they climbed even more quickly than the bishop and knight do, yielding no new insights.

6.2.1 Tuning from Arbitrary Values

Figure 1 illustrates the learning. The 11 parameters were all initialized to 50, where 100 represents both the value of a pawn and the default value of most scaling factors. For ease of interpretation, legend contents are ordered to match up with the vertical ordering of corresponding data at the rightmost point on the x-axis. For instance, bishop is the topmost value, followed by knight, then τ, and so on. τ is measured on the left y-axis in linear scale; weights are measured on the right y-axis in logarithmic scale, for improved visibility of the weight trajectories.

Rapid improvement is made as the bishop and knight weights climb swiftly to about 285, after which τ continues to climb, albeit more slowly. We attribute most of the improvement in τ to the proper determination of weight values for the minor pieces. All the material and positional weights are tuned to reasonable values.
The scaling factors learned are more interesting. The king tropism and pawn structure scaling factors gradually reached, then exceeded CRAFTY’s default values of 100. The scaling factors for blocked pawns, passed pawns, and king safety are lower, but not unreasonably so. However, the king safety asymmetry scaling factor dives quickly and relentlessly. CRAFTY’s default value for this term is −40; perhaps we should have started it at a lower value to speed convergence.

Tables 6 and 7 contain match results of the weight vectors at specified iterations during the learning illustrated in Figure 1. Each side plays each starting position both as White and as Black, so with the Nunn 30 test, 60 games are played, and with the Dailey 200 test, 400 games are played. Games reaching move 121 were declared drawn.

The play of the tuned program improves dramatically as learning occurs. Of interest is the apparent gradual decline in percentage score for later iterations on the Nunn 30 test suite. The DEEP THOUGHT team (Nowatzyk, 2000) found that their best parameter settings were achieved before reaching maximum agreement with GM players. Perhaps we are also experiencing this phenomenon. We used the Dailey 200 test suite to attempt to confirm
that this was a real effect, and found that by this measure too, the weight
vectors at iterations 300 and 400 were superior to later ones.

<table>
<thead>
<tr>
<th>iteration</th>
<th>wins</th>
<th>draws</th>
<th>losses</th>
<th>percentage score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1</td>
<td>56</td>
<td>5.83</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>9</td>
<td>48</td>
<td>12.50</td>
</tr>
<tr>
<td>200</td>
<td>14</td>
<td>21</td>
<td>25</td>
<td>40.83</td>
</tr>
<tr>
<td>300</td>
<td>21</td>
<td>26</td>
<td>13</td>
<td>56.67</td>
</tr>
<tr>
<td>400</td>
<td>19</td>
<td>28</td>
<td>13</td>
<td>55.00</td>
</tr>
<tr>
<td>500</td>
<td>18</td>
<td>26</td>
<td>16</td>
<td>51.67</td>
</tr>
<tr>
<td>600</td>
<td>18</td>
<td>23</td>
<td>19</td>
<td>49.17</td>
</tr>
</tbody>
</table>

Table 6. Match results (11 weights tuned from 50 vs. default weights), 5 minutes per game, Nunn 30 test suite.

Throughout our experimentation, we have found that our tuned feature weights tend to perform better on the Nunn test suite than the Dailey test suite. Nunn’s suite contains positions of particular strategic and tactical complexity. Dailey’s suite is largely more staid, and contains positions from much earlier in the game. CRAFTY’s default weights appear to be more comfortable with the latter than the former.

<table>
<thead>
<tr>
<th>iteration</th>
<th>wins</th>
<th>draws</th>
<th>losses</th>
<th>percentage score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>13</td>
<td>384</td>
<td>2.38</td>
</tr>
<tr>
<td>100</td>
<td>12</td>
<td>31</td>
<td>357</td>
<td>6.88</td>
</tr>
<tr>
<td>200</td>
<td>76</td>
<td>128</td>
<td>196</td>
<td>35.00</td>
</tr>
<tr>
<td>300</td>
<td>128</td>
<td>152</td>
<td>120</td>
<td>51.00</td>
</tr>
<tr>
<td>400</td>
<td>129</td>
<td>143</td>
<td>128</td>
<td>50.13</td>
</tr>
<tr>
<td>500</td>
<td>107</td>
<td>143</td>
<td>150</td>
<td>44.63</td>
</tr>
<tr>
<td>600</td>
<td>119</td>
<td>158</td>
<td>123</td>
<td>49.50</td>
</tr>
</tbody>
</table>

Table 7. Match results (11 weights tuned from 50 vs. default weights), 5 minutes per game, Dailey 200 test suite.

6.2.2 Tuning from CRAFTY’s Default Values

We repeated the just-discussed experiment with one change: the feature weights start at CRAFTY’s default values rather than at 50. Figure 2 depicts the learning. Note that we have negated the values of the king safety asymmetry scaling factor in the graph so that we could retain the logarithmic scale on the right y-axis, and also for another reason, for which see below.

While most values remain normal, the king safety scaling factor surprisingly rises to almost four times the default value. Meanwhile, the king safety asymmetry scaling factor descends even below −100. The combination indicates a complete lack of regard for the opponent’s king safety, but great regard for its own. Table 8 shows that this conservative strategy is by no means an improvement.

We conclude that the learning is able to yield settings that perform comparably to settings tuned by hand over years of games versus grandmasters.
The most unusual behaviour of the king safety and king safety asymmetry scaling factors deserves specific attention. When the other nine terms are left constant, these two terms behave similarly to how they do when all eleven terms are tuned. In contrast, when these two terms are held constant, no statistically significant performance difference is found between the learned weights and CRAFTY’s default weights. When the values of the king safety asymmetry scaling factor are negated as in Figure 2, it becomes visually clear from their trajectories that the two terms are behaving in a codependent manner. More investigation is required to determine the root cause of this behaviour.
7. Conclusion

We have proposed a new procedure for optimizing static evaluation functions based upon globally ordering a multiplicity of positions in a consistent manner. This application of ordinal correlation is fundamentally different from prior evaluation function tuning techniques. We believe it is worth further exploration, and hope it will lead to a new perspective and fresh insights about decision making in game-tree search.

While our initial results show promise, more work is certainly needed. It is important to keep in mind that we tuned feature weights in accordance with human assessments. Doing so may simply not be optimal for computer play. Nonetheless, it is worth noting that having reduced the playing ability of a grandmaster-level program to candidate master strength by significantly altering several important feature weights, the learning algorithm was able to restore the program to grandmaster strength.

7.1 Reflection

Having identified the anomalous behaviour in Figure 2, it is worth looking again at Figure 1. The match results suggest that all productive learning occurred by iteration 400 at the latest, after which a small but perceptible decline appears to occur. The undesirable codependency between the king safety and king safety asymmetry scaling factors also appears to be present in the later iterations of the first experiment.

Furthermore, our training data is small enough \((n = 32,768)\) that overfitting is a consideration. Future learning experiments should use more positions. This may in turn reduce the search effort required per position to tune weights well. Although we are not certain why larger searches improve the quality of learning, as the amount of search used per machine assessment increases, the amount of information gathered about how relative weights interact also increases. On the surface, then, the improvement is not illogical.

While some weights, for instance the positional rook terms, learned nearly identical values in both experiments, other features exhibited more variance. For cases such as the king tropism and blocked pawns scaling factors, it could be that comparable performance may be achieved with a relatively wide range of values.

In our reported experiments, computation of \(\tau\) was dominated by the search effort to generate machine assessments, enough so that the use of Spearman’s \(\rho\) (or perhaps even Pearson correlation, notwithstanding our original rationale) may also have been possible. Maximizing these alternative metrics could be tried, at least when the training data contains
relatively few positions. Other optimization strategies, for instance genetic algorithms, could also be tried.

It was not originally planned to attempt to maximize $\tau$ only upon assessments at a specific level of search effort. Unfortunately, we encountered implementation difficulties, and so reverted to the approach described herein. We had intended to log the node number or time point along with the new score whenever the evaluation of a position changes. This would have, without the use of excessive storage, provided the precise score at any point throughout the search. We would have tuned to maximize the integral of $\tau$ over the period of search effort. Implementation of this algorithm would more explicitly reward reaching better evaluations more quickly, improving the likelihood of tuning feature weights and perhaps even search control parameters effectively.

7.2 Future Directions

While our experiments used chess assessments from humans, it is possible to use assessments from deeper searches and/or from a stronger engine, or to tune a static evaluation function for a different domain. Depending on the circumstances, merging consecutively-ordered fine-grained assessments into fewer, larger categories may be desirable. Doing so could even become necessary should the computation of $\tau$ dominate the time per iteration, but this is unlikely unless one uses only negligible search to form machine assessments.

Elidan et al. (2002) found that perturbation of training data could assist in escaping local maxima during learning. Our implementation of $\tau$, designed with this finding in mind, allows non-integer weights to be assigned to each cell. Perturbing the weights in an adversarial manner as local maxima are reached, so that positions are weighted slightly more important when generally discordant, and slightly less important when generally concordant, could allow the learner to continue making progress.

It would also be worthwhile to examine positions of maximum disagreement between human and machine assessments, in the hope that study of the resulting positions will identify new features that are not currently present in CRAFTY’s evaluation. Via this process, a number of labeling errors would be identified and corrected. However, we do not believe that this would materially affect the outcome of the learning process.

A popular pastime amongst computer chess hobbyists is to attempt to discover feature weight settings that result in play mimicking their favourite human players. By tuning against appropriate training data, e.g., from opening monographs and analyses published in *Chess Informant* and elsewhere that are authored by the player to be mimicked, training an
evaluation function to assess positions similarly to how a particular player might actually do so should now be possible.

Producers of top computer chess software play many games against their commercial competitors. They could use our method to model their opponent’s evaluation function, then use this model in a minimax (no longer negamax) search. Matches then played would be more likely to reach positions where the two evaluation functions differ most, providing improved winning chances for the program whose evaluation function is more accurate, and object lessons for the subsequent improvement of the other.

Identifying the most realistic mapping of CRAFTY’s machine assessments to the seven human positional assessments is also of interest. This information would allow CRAFTY (or a graphical user interface connected to CRAFTY) to present scoring information in a human-friendly format alongside the machine score.

Acknowledgements

We would like to thank: Yngvi Björnsson, for the use of his automated game-playing software, and for fruitful discussions; Don Dailey, for access to his suite of 200 test positions; Robert Hyatt, for making CRAFTY available, and also answering questions about its implementation; Peter McKenzie, for providing PGN to EPD conversion software; NSERC, for partial financial support [Grant OPG 7902 (Marsland)].

References


FIRST EXPERIMENTAL RESULTS OF PROBCUT APPLIED TO CHESS

A.X. Jiang
Department of Computer Science, University of British Columbia, Vancouver, Canada
albertjiang@yahoo.com

M. Buro
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
mburo@cs.ualberta.ca, http://www.cs.ualberta.ca/~mburo/

Abstract
ProbCut is a selective-search enhancement to the standard alpha–beta algorithm for two–person games. ProbCut and its improved variant Multi–ProbCut (MPC) have been shown to be effective in Othello and Shogi, but there had not been any report of success in the game of chess previously. This paper discusses our implementation of ProbCut and MPC in the chess engine CRAFTY. Initial test results suggest that the MPC version of CRAFTY is stronger than the original version of CRAFTY: it searches deeper in promising lines and defeated the original CRAFTY +22−10 = 32 (59.4%) in a 64–game match. Incorporating MPC into CRAFTY also increased its tournament performance against YACE – another strong chess program: CRAFTY’s speed chess tournament score went up from 51% to 56%.

Keywords: Selective search, ProbCut, chess

1. Introduction

Computer chess has been an AI research topic since the invention of the computer, and it has come a long way. Nowadays, the best computer chess programs and the best human grandmasters play at roughly the same level. Most of the successful chess programs use the so–called brute–force approach, in which the program has limited chess knowledge and relies on a fast search algorithm to find the best move. There has been much research on improving the original minimax algorithm for finding moves in two player perfect information games. Enhancements range from sound backward pruning (alpha–beta search), over using transposition tables and iterative deepening, to selective search heuristics
that either extend interesting lines of play or prune uninteresting parts of the search tree.

The ProbCut (Buro, 1995) and Multi–ProbCut (MPC) (Buro, 1997a) heuristics fall into the last category. They were first implemented in Othello programs where they resulted in a much better performance compared to full–width alpha–beta search. Utilizing MPC, Logistello defeated the reigning human Othello World Champion Takeshi Murakami by a score of 6–0 in 1997 (Buro, 1997b).

ProbCut and MPC do not rely on any game specific properties. However, there were no previous reports of success at implementing them in the game of chess. In this paper we present our first implementations of ProbCut and MPC in a chess program and some experimental results on their performance. Section 2 gives some necessary background knowledge. Section 3 discusses our ProbCut implementation and Section 4 discusses our MPC implementation. Finally, Section 5 concludes and discusses some ideas for future research.

2. Background

There has been a lot of previous research in the field of game–tree search. We will not attempt to cover it all here. Instead, we will concentrate on things relevant to ProbCut. For an introduction to game–tree search, a good web–site is www.xs4all.nl/~verhelst/chess/search.html.

2.1 Minimax and Alpha–Beta Search

For two–person zero–sum games like chess, positions can be viewed as nodes in a tree or DAG. In this model, moves are represented by edges which connect nodes. Finding the best move in a given positions then means to search through the successors of the position in order to find the best successor for the player to move after finding the best successor for the opponent in the next level of the tree. This procedure is called minimaxing. In practice, computers do not have time to search to the end of the game. Instead, they search to a certain depth, and use a heuristic evaluation function to evaluate the leaf nodes statically. For chess, the evaluation function is based on material and other considerations such as king safety, mobility, and pawn structure.

An important improvement over minimax search is alpha–beta pruning (Knuth and Moore, 1975). An alpha–beta search procedure takes additional parameters alpha and beta, and returns the correct minimax value (up to a certain depth) if the value is inside the window (alpha, beta). A returned value greater or equal to beta is a lower bound on the the minimax value, and a value less or equal to alpha is an upper bound. These cases are called fail–high and fail–low, respectively. A pseudo–code representation of one version of the algorithm is shown in Figure 1. The algorithm shown is called “fail–hard” alpha–beta, because it generally returns alpha for fail–lows and beta for fail–highs. There
First Experimental Results of ProbCut Applied to Chess

```c
int AlphaBeta(int alpha, int beta, int height) {
    if (height == 0) return Evaluation();

    int total_moves = GenerateMoves();
    for (int i=0; i < total_moves; i++) {
        MakeMove(i);
        val = -AlphaBeta(-beta, -alpha, height-1);
        UndoMove(i);
        if (val >= beta) return val;
        if (val > alpha) alpha = val;
    }
    return alpha;
}
```

Figure 1. The alpha-beta algorithm (fail-hard version).

exist "fail-soft" versions of alpha-beta which can return values outside of the alpha-beta window, thus giving better bounds when it fail-high/fail-low.

There have been a number of enhancements to alpha-beta, e.g. transposition tables, iterative deepening, NegaScout, etc. (Reinefeld, 1983; Junghanns, 1998). Armed with these refinements, alpha-beta has become the dominant algorithm for game tree searching (Junghanns, 1998).

Compared to minimax, alpha-beta is able to prune many subtrees that would not influence the minimax value of the root position. But it still spends most of its time calculating irrelevant branches that human experts would never consider. Researchers have been trying to make the search more selective, while not overlooking important branches. How should we decide whether to search a particular branch or not? One idea is to base this decision on the result of a shallower search. The null-move heuristic (Beal, 1990; Donninger, 1993) and ProbCut are two approaches based on this idea.

2.2 The Null-Move Heuristic

A null-move is equivalent to a pass: the player does nothing and lets the opponent move. Passing is not allowed in chess, but in chess games it is almost always better to play a move than passing. The null-move heuristic (or null-move pruning) takes advantage of this fact, and before searching the regular moves for height-1 plies as in alpha-beta, it does a shallower search on the null-move for height-\( R - 1 \) plies, where \( R \) is usually 2. If the search on the null-move returns a value greater or equal to beta, then it is very likely that one of the regular moves will also fail-high. In this case we simply return beta after the search on the null-move. This procedure can even be applied recursively in the shallower search, as long as no two null-moves are played consecutively.

Because the search on the null-move is shallower than the rest, occasionally it will overlook something and mistakenly cut the branch, but the speed-up from
cutting these branches allows it to search deeper on more relevant branches. The benefits far outweigh the occasional mistakes. However, in chess endgames with few pieces left, zugzwang positions are often encountered, in which any move will deteriorate the position. Null-move heuristic fails badly in zugzwang positions. As a result, chess programs turn off null-move heuristic in late endgames.

There have been some research to further fine-tune and improve the null-move heuristic. Adaptive Null-Move Pruning (Heinz, 1999) uses $R = 3$ for positions near the root of the tree and $R = 2$ for positions near the leaves of the tree, as a compromise between the too aggressive $R = 3$ and the robust but slower $R = 2$. Verified Null-Move Pruning (Tabibi and Netanyahu, 2002) uses $R = 3$, but whenever the shallow null-move search returns a fail-high, instead of cutting, the search is continued with reduced depth. Verified null-move pruning can detect zugzwang positions, have better tactical strength while searching less nodes than standard $R = 2$.

The null-move heuristic is very effective in chess, and most of the strong chess engines use it. But it depends on the property that the right to move has positive value, so it is not useful to games like Othello and checkers, in which zugzwang positions are common.

2.3 ProbCut

ProbCut is based on the idea that the result $v'$ of a shallow search is a rough estimate of the result $v$ of a deeper search. The simplest way to model this relationship is by means of a linear model:

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

where $e$ is a normally distributed error variable with mean 0 and standard deviation $\sigma$. The parameters $a$, $b$, and $\sigma$ can be computed by linear regression applied to the search results of thousands of positions.

If based on the value of $v'$, we are certain that $v \geq \beta$, where $\beta$ is the beta-bound for the search on the current subtree, we can prune the subtree and return $\beta$. After some algebraic manipulations, the above condition becomes

$$(av' + b - \beta) / \sigma \geq -e / \sigma.$$ 

This means that $v \geq \beta$ holds true with probability of at least $p$ iff $(av' + b - \beta) / \sigma \geq \Phi^{-1}(p)$. Here, $\Phi$ is the standard Normal distribution. This inequality is equivalent to $v' \geq (\Phi^{-1}(p) \cdot \sigma + \beta - b) / a$. Similarly for $v \leq \alpha$, the condition becomes

$$v' \leq (\Phi^{-1}(p) \cdot \sigma + \alpha - b) / a.$$ 

This leads to the pseudo-code implementation shown on Figure 2. Note that the search windows for the shallow searches are set to have width 1. These are called null-window searches. Generally, the narrower the window is, the earlier the search returns. Null-window searches are very efficient when we do not care about the exact minimax value and only want to know whether the value is above or below a certain bound, which is the case here. The depth pair and
cut threshold are to be determined empirically, by checking the performance of
the program with various parameter settings.

For ProbCut to be successful, \( v' \) needs to be a good estimator of \( v \), with a fairly small \( \sigma \). This means that the evaluation function needs to be a fairly accurate estimator of the search results. Evaluation functions for chess are generally not very accurate, due to opportunities of capturing which cannot be resolved statically. Fortunately, most chess programs conduct a so-called quiescence search: at the leaves of the game tree where the regular search height reaches zero, instead of calling the evaluation function, a special quiescence search function is called to search only capturing moves, only using the evaluation function's results when there are no profitable capturing moves. Quiescence search returns a much more accurate value.

In summary, the null-move heuristic and ProbCut both try to compensate for the lower accuracy of the shallow search by making it harder for the shallow search to produce a cut. The null-move heuristic does this by giving the opponent a free move, while ProbCut widens the alpha-beta window.

```c
#define S 4 // depth of shallow search
#define H 8 // check height
#define T 1.0 // cut threshold

int AlphaBeta(int alpha, int beta, int height) {
    if (height == 0) return Evaluation();

    if (height == H) {
        int bound;

        // is v >= beta likely?
        bound = round ((T * sigma + beta - b) / a);
        if (AlphaBeta(bound-1, bound, S) >= bound)
            return beta;

        // is v <= alpha likely?
        bound = round ((-T * sigma + alpha - b) / a);
        if (AlphaBeta(bound, bound+1, S) <= bound)
            return alpha;
    }

    // The rest of alpha-beta code goes here
    ...
}
```

*Figure 2.* ProbCut implementation with depth pair (4,8) and cut threshold 1.0.
2.4 Multi–ProbCut

MPC enhances ProbCut in several ways:

- Allowing different regression parameters and cut thresholds for different stages of the game.
- Using more than one depth pair. For example, when using depth pairs (3,5) and (4,8), if at check height 8 the 4-ply shallow search does not produce a cut, then further down the 8-ply subtree we could still cut some 5-ply subtrees using 3-ply searches.
- Internal iterative deepening for shallow searches.

Figure 3 shows pseudo-code for a generic implementation of MPC. The MPC search function is not recursive in the sense that ProbCut is not applied inside the shallow searches. This is done to avoid the collapsing of search depth. In the case of Othello, MPC shows significant improvements over ProbCut.

2.5 ProbCut and Chess

There has been no report of success for ProbCut or MPC in chess thus far. There are at least two reasons for this:

1. The null-move heuristic has been successfully applied to chess. Null-move and ProbCut are based on similar ideas. As a result they tend to prune the same type of positions. Part of the reason why ProbCut is so successful in Othello is that the null-move heuristic does not work in Othello because it is a zugzwang game. But in chess, ProbCut and MPC have to compete with null-moves, which already improves upon brute-force alpha-beta search.

2. The probability of a chess search making a serious error is relatively high, probably due to the higher branching factor (Junghanns et al., 1997). This leads to a relatively large standard deviation in the linear relationship between shallow and deep search results, which makes it harder for ProbCut to prune sub-trees.

In the GAMES group at the University of Alberta there had been attempts to make ProbCut work in chess in 1997 (Junghanns and Brockington, 2002). However, the cut-thresholds were chosen too conservatively resulting in a weak performance.

Recently, researchers in Japan have successfully applied ProbCut to Shogi (Shibahara, Inui, and Kotani, 2002). In Shogi programs forward pruning methods are not widely used, because Shogi endgames are much more volatile than chess endings. Therefore, ProbCut by itself can easily improve search performance compared with plain alpha-beta searchers. As mentioned above, gaining improvements in chess, however, is much harder because of the already very good performance of the null-move heuristic.
#define MAX_STAGE 2 // e.g. middle-game, endgame
#define MAX_HEIGHT 10 // max. check height
#define NUM_TRY 2 // max. number of checks

// ProbCut parameter sets for each stage and height

struct Param {
  int d; // shallow depth
  float t; // cut threshold
  float a, b, s; // slope, offset, std.dev.
} param[MAX_STAGE+1][MAX_HEIGHT+1][NUM_TRY];

int MPC(int alpha, int beta, int height) {

  // ProbCut check
  if (height <= MAX_HEIGHT) {
    for (int i=0; i < NUM_TRY; i++) {
      int bound;
      Param &pa = param[stage][height][i];

      // skip if there are no parameters available
      if (pa.d < 0) break;

      // is v_height >= beta likely?
      bound = round((pa.t*pa.s+beta-pa.b)/pa.a);
      if (AlphaBeta(bound-1, bound, pa.d) >= bound)
        return beta;

      // is v_height <= alpha likely?
      bound = round((-pa.t*pa.s+alpha-pa.b)/pa.a);
      if (AlphaBeta(bound, bound+1, pa.d) <= bound)
        return alpha;
    }
  }

  // the remainder of the alpha-beta algorithm
  ...
}

Figure 3. Multi-ProbCut implementation. AlphaBeta() is the original alpha–beta search function.

3. ProbCut Implementation

Before trying MPC, we implemented the simpler ProbCut heuristic with one depth pair and incorporated it into CRAFTY (version 18.15) by Hyatt.¹

¹CRAFTY's source code is available at ftp://ftp.cis.uab.edu/pub/hyatt.
CRAFTY is a state-of-the-art free chess engine. It uses a typical brute-force approach, with a fast evaluation function, NegaScout search and all the standard search and all the standard enhancements: transposition table, Null-Move heuristic, etc. CRAFTY also utilizes quiescence search, so the results of its evaluation function plus quiescence search are fairly accurate.

The philosophy of our approach is to take advantage of the speed-up provided by the null-move heuristic whenever possible. One obvious way to combine the null-move and ProbCut heuristics is to view null-move search as part of the brute-force search, and build ProbCut on top of the "alpha-beta plus null-move" search. Applying the necessary changes to CRAFTY is easy. We put the ProbCut shallow search code in front of the null-move shallow search code. We also implemented the MPC feature that allows different parameters to be used for middle-game and endgame.

Before ProbCut-CRAFTY could be tested, parameters of the linear ProbCut opinion change model had to be estimated. We let CRAFTY search (using alpha-beta with null-move heuristic) around 2700 positions and record its search results for 1, 2, ..., 10 plies. The positions were chosen randomly from some computer chess tournament games and some of CRAFTY's games against human grandmasters on internet chess servers. Note that that CRAFTY was using the null-move heuristic for these searches.

Then we fitted the linear regression model for several depth pairs and game phases, using the data collected. The results indicate that shallow and deep search results are correlated, as shown in Figure 4. However, the fit is not perfect. The \( v' \) versus \( v \) relation has the following characteristics.

- The slope is closer to 1.0 and the standard deviation smaller for \( v' \) data points closer to zero. For example, for depth pair (4, 8), and \( v' \) data points in the range \([-300, 300]\), the slope is 1.07 and the standard deviation is 83; for \( v' \) data points in the range \([-1000, 1000]\), the slope is 1.13 and the standard deviation is 103. This can be explained as follows: if say White has a big advantage, then White will likely gain more material advantage after a few more moves. Therefore, if the shallow search returns a big advantage, a deeper search will likely return a bigger advantage, and vice versa for disadvantages. We only used \( v' \) data points in the range \([-300, 300]\) for the linear regression.

- Occasionally the shallow search misses a check-mate while the deeper search finds it. For example, in a position White can check-mate in 7 plies. A 4-ply search cannot find the check-mate while a 8-ply search can find it. For the depth pair (4, 8), and \( v' \) data points in the range \([-300, 300]\), this happens roughly once every 1000 positions. A check-mate-in-\( N \)-moves is represented by a large integer in CRAFTY. We excluded these data points from the linear regression, because the evalu-
First Experimental Results of ProbCut Applied to Chess

Figure 4. \( v' \) versus \( v \) for depth pair (4,8) The evaluation function’s scale is 100 = one pawn, i.e. a score of 100 means the player to move is one pawn up (or has equivalent positional advantage).

<table>
<thead>
<tr>
<th>Pairs</th>
<th>Stage</th>
<th>( a )</th>
<th>( b )</th>
<th>( \sigma )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,5)</td>
<td>middle–game</td>
<td>0.998</td>
<td>-7</td>
<td>55.8</td>
<td>0.90</td>
</tr>
<tr>
<td>(3,5)</td>
<td>endgame</td>
<td>1.026</td>
<td>-4.1</td>
<td>51.8</td>
<td>0.94</td>
</tr>
<tr>
<td>(4,8)</td>
<td>middle–game</td>
<td>1.02</td>
<td>2.36</td>
<td>82</td>
<td>0.82</td>
</tr>
<tr>
<td>(4,8)</td>
<td>endgame</td>
<td>1.11</td>
<td>1.75</td>
<td>75</td>
<td>0.90</td>
</tr>
</tbody>
</table>

Table 1. Linear regression results. The evaluation function’s scale is 100 = one pawn. \( r \) is the regression correlation coefficient, a measure of how good the data fits the linear model.

...of check–mate is a rather arbitrary large number, there is no proper way to incorporate these data points in the linear regression.

We also fitted model parameters for different game stages. It turned out that the standard deviation for the fit using only endgame positions\(^2\) is smaller than the standard deviation using only middle–game positions. Table 1 shows some of the results.

We conducted some experiments\(^3\) with different depth pairs and cut thresholds. Depth pairs (4, 6) and (4, 8), and cut thresholds 1.0 and 1.5 were tried.

\(^2\)In CRAFTY endgame positions are defined as those in which both players have weighted material count less than 15. Here Queen is 9, Rook is 5, Knight/Bishop is 3, and Pawns do not count.

\(^3\)All initial experiments were run on Pentium–3/850MHz and Athlon–MP/1.66GHz machines under Linux, whereas the later tournaments were all played on Athlon–MP/2GHz machines. CRAFTY’s hash table size
We used two types of tests. First, we test the search speed by running fixed-time searches and look at the depths reached. If a ProbCut version is not faster than the plain null-move version, then the ProbCut version is clearly no good. If a ProbCut version is faster than null-move, it is still not necessarily better. So to test the overall performance, we then run matches between the promising ProbCut versions and the original CRAFTY.

We let the program search about 300 real-game positions, spending 30 seconds on each position, and see how deep it was able to search on average. Results show that

- Versions with depth pairs (4,6) and (4,8) have similar speeds.
- The versions with cut threshold 1.5 are not faster than plain CRAFTY.
- The versions with cut threshold 1.0 are slightly faster than CRAFTY: they search 11.6 plies compared to 11.5 plies by CRAFTY. In some positions, 80 – 90% of the shallow searches result in cuts, and ProbCut is much faster than plain CRAFTY. But in some other positions the shallow searches produce cuts less than 60% of the time, and ProbCut is about the same speed or even slower than CRAFTY. On average, this version of ProbCut produces more cuts than plain CRAFTY’s null-move heuristic does at the check height.

Because the cut threshold 1.5 is no good, we concentrated on the threshold 1.0 for the following experiments. We ran matches between the ProbCut versions and plain CRAFTY. Each side has 10 minutes per game. A generic opening book was used. Endgame databases were not used. A conservative statistical test\(^4\) shows that in a 64-game match, a score above 38 points (or 59%) is statistically significant with \(p < 0.05\). Here a win counts one point and a draw counts half a point.

The match results are not statistically significant. The ProbCut versions seem to be no better nor worse than plain CRAFTY. For comparison, we ran a 64–game match of ProbCut against CRAFTY with null–move turned off for both programs. The ProbCut version is significantly better than CRAFTY here, winning the match 40–24.

4. **Multi–ProbCut Implementation and Results**

ProbCut produces more cuts than the plain null-move heuristic does, but it seems that the small speed-up provided by ProbCut is not enough to result

---

\(^4\)The statistical test is based on the assumption that at least 30% of chess games between these programs are draws, which is a fair estimate. The test is based on Amir Ban’s program from his posting on rec.game.chess.computer:

http://groups.google.com/groups?hl=en&lr=&ie=UTF-8&selm=33071608.796A%40msys.co.il
Table 2.  Endgame threshold optimization results.  Reported are the point percentages for MPC–CRAFTY playing 64-game tournaments against CRAFTY using different values for the endgame cut thresholds.  Game timing was 2 minutes per player per game plus 12 seconds increment on an Athlon-MP 1.67 GHz.  The middle-game threshold was fixed at 1.0.

<table>
<thead>
<tr>
<th>$t_{end}$</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
<th>1.0</th>
<th>1.05</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC %</td>
<td>53.9</td>
<td>59.3</td>
<td>53.1</td>
<td>48.5</td>
<td>51.6</td>
<td>57.8</td>
<td>52.3</td>
<td>54.7</td>
<td>51.6</td>
<td>51.6</td>
</tr>
</tbody>
</table>

Table 3.  Middle-game threshold optimization results.  With the endgame threshold fixed at 1.0 we repeated the 64-game tournaments now using faster hardware (Athlon–MP 2 GHz) that just became available and longer time controls: 10 minutes per player per game plus 60 seconds increment.  Each tournament took about eight CPU days.

<table>
<thead>
<tr>
<th>$t_{mid}$</th>
<th>0.8</th>
<th>0.9</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC %</td>
<td>54.7</td>
<td>59.4</td>
<td>57.8</td>
<td>58.6</td>
<td>59.4</td>
<td>53.1</td>
</tr>
</tbody>
</table>

in better playing strength.  This motivates our implementation of MPC.  We already have different regression parameters for middle–game and endgame in our ProbCut implementation.  Now we implemented multiple depth pairs.  The implementation was straightforward, much like the pseudo-code in Figure 3.

After initial experiments which showed that the null–move heuristic excels at small heights, we chose depth pairs (2,6), (3,7), (4,8), (3,9), and (4,10) for endgames and middle–games.  Another reason for choosing pairs with increasing depth differences is that otherwise the advantage of MPC rapidly diminishes in longer timed games.  We tested the speed of the MPC implementation using a cut threshold of 1.0 on the same 300+ positions as in Section 1.3.  With 30 seconds per position, it is able to search 12.0 plies on average, which is 0.5 plies deeper than original CRAFTY.

For optimizing the endgame and middle–game cut thresholds we then ran two sets of 64–game tournaments between MPC–CRAFTY and the original version.  In the first phase we kept the middle–game cut threshold fixed at 1.0 and varied the endgame threshold.  The results shown in Table 2 roughly indicate good threshold choices.  However, the high fluctuations suggest that we should play more games to get better playing strength estimates.  After some more experimentation we fixed the endgame threshold at 1.0 and went on to optimizing the middle–game cut threshold by playing a second set of tournaments, now on faster hardware and longer time controls.  Threshold pairs (1.2, 1.0) and (1.0, 1.0) resulted in the highest score (59.4%) against the original CRAFTY version.

In order to validate the self–play optimization results, we let MPC–CRAFTY play a set of tournaments against YACE – a strong chess program written by Dieter Buerssner, which is available for Linux and can be downloaded from http://home1.stofanet.dk/moq/.  Table 4 summarizes the promising re-
sults which indicate a moderate playing strength increase even against other chess programs when using MPC.

<table>
<thead>
<tr>
<th>Pairing</th>
<th>CRAFTY % (2min+10sec/move)</th>
<th>CRAFTY % (8min+20sec/move)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRAFTY vs. YACE</td>
<td>42.0%</td>
<td>50.8%</td>
</tr>
<tr>
<td>MPC–CRAFTY (1.2,1.0) vs. YACE</td>
<td>53.1%</td>
<td>56.3%</td>
</tr>
<tr>
<td>MPC–CRAFTY (1.0,1.0) vs. YACE</td>
<td>57.0%</td>
<td>55.5%</td>
</tr>
</tbody>
</table>

Table 4. Results of 64-game tournaments played by three CRAFTY versions against YACE using two different time controls.

5. Conclusions and Further Research

Preliminary results show that MPC can be successfully applied to chess. Our MPC implementation shows clear improvement over our ProbCut (plus variable parameters for different stages) implementation. This indicates that the main source of improvement in MPC is the use of multiple depth pairs. Due to the already good performance of the null–move heuristic in chess, the improvement provided by MPC in chess is not as huge as in Othello. However, our implementation, which combines MPC and null–move heuristic, shows definite advantage over the plain null–move heuristic in CRAFTY, as shown by the match results. MPC is relatively easy to implement. We encourage chess programmers to try MPC in their chess programs.

More experiments need to be conducted on our MPC implementation to determine how evaluation function parameters like the king safety weight can influence MPC’s performance. To further verify the strength of the MPC implementation, we plan to run matches with even longer time controls.

The depth pairs and the cut threshold can be further fine–tuned. One way to optimize them is to run matches between versions with different parameters. But better results against another version of the same program do not necessarily translate into better results against other opponents. An alternative would be to measure the accuracy of search algorithms by a method similar to the one employed in (Junghanns et al., 1997), using a deeper search as the “oracle,” and looking at the difference between the oracle’s evaluations on the oracle’s best move and the move chosen by the search function we are measuring. Maybe the combination of the above two methods gives a better indication of chess strength.
First Experimental Results of ProbCut Applied to Chess

Acknowledgements

We would like to thank David Poole for his helpful comments, and Bob Hyatt for making the source code of his excellent and very readable CRAFTY chess program available to the public.

References


---

\(^5\)The author's articles can be downloaded for personal use from http://www.cs.ualberta.ca/~mburo/publications.html
SEARCH VERSUS KNOWLEDGE: AN EMPIRICAL STUDY OF MINIMAX ON KRK

A. Sadikov, I. Bratko, I. Kononenko
*University of Ljubljana, Faculty of Computer and Information Science, Tržaška 25, 1000 Ljubljana, Slovenia*
aleksander.sadikov@fri.uni-lj.si

**Abstract**

This article presents the results of an empirical experiment designed to gain insight into what is the effect of the minimax algorithm on the evaluation function. The experiment's simulations were performed upon the KRK chess endgame. Our results show that dependencies between evaluations of sibling nodes in a game tree and an abundance of possibilities to commit blunders present in the KRK endgame are not sufficient to explain the success of the minimax principle in practical game-playing as was previously believed. The article shows that minimax in combination with a noisy evaluation function introduces a bias into the backed-up evaluations and argues that this bias is what masked the effectiveness of the minimax in previous studies.

**Keywords:** minimax principle, KRK chess endgame, evaluation-function quality, bias

1. **Introduction**

Over twenty years ago Beal (1980) set out to analyze whether and why values backed up from minimax search are more trustworthy than the heuristic values themselves. He constructed a simple mathematical model to analyze the minimax algorithm. To his surprise the analysis of the model showed that the backed-up values were actually somewhat less trustworthy than the heuristic values themselves. He then wrote: “This result is disappointing. It was hoped that the analysis would show that the probability of error reduced with backing-up.” A couple of years later two articles (Beal, 1982; Bratko and Gams, 1982) simultaneously conducted further analysis into why minimax does yield good results in practical game-playing while apparently backed-up values seem less reliable; both articles reached the same conclusion. They argued that the true values of sibling nodes in a game tree are not independent of one another. This clustering of similar values is a
major feature in practical games and it was this phenomenon that Beal’s mathematical model did not account for. The problem with the minimax paradigm under the assumption of independence of sibling values was also confirmed by Nau (1982, 1983), who called this a search-depth pathology in game trees. In a simulation Nau (1982) introduced strong dependencies between sibling nodes and discovered that this can cause search-depth pathology to disappear.

However, Pearl (1984) partly disagreed with the conclusion reached by Beal, Bratko, Gams and Nau, and claimed that while strong dependencies between sibling nodes in a game tree can eliminate the pathology, practical games like chess do not possess dependencies of sufficient strength. He pointed out that few chess positions are so strong that they cannot be spoiled abruptly if one really tries hard to do so. He concluded that the success of minimax in game-playing programs is “based on the fact that common games do not possess a uniform structure but are riddled with early terminal positions, colloquially named blunders, pitfalls or traps. Close ancestors of such traps carry more reliable evaluations than the rest of the nodes, and when more of these ancestors are exposed by the search, the decisions become more valid.” Moreover, Schrüfer (1986) and its follow-up (Althöfer, 1989) did some further analysis of pathology in game trees. Especially interesting is their observation that to avoid pathology, an evaluation function must, among other things, have negligible probability of underestimating a position from the perspective of the player to move.

All of the above studies have two things in common: (a) they accept the empirical evidence that the minimax principle works in practical game-playing programs and (b) they try to model mathematically the minimax algorithm and theoretically deduce what happens when heuristic values assigned to leaves are backed-up towards the root of the game tree. To make such mathematical analysis feasible the researchers are forced to make certain assumptions about the game they model and to make simplifications in their model. Thus, the results of these models are always to be viewed with the acknowledgement of this assumptions and simplifications in the back of one’s mind. In contrast to that, our approach in this article is to take (part of) a real game with a real evaluation function and observe empirically what is going on when we change the search depth and the quality of the evaluation function. We have at our disposal an absolutely correct evaluation function which we can corrupt in a controlled way. We also have a minimax search engine that is capable of searching to very high depths because of its efficient implementation.

The next section describes our choice of the game, the evaluation function and its artificial corruption, as well as the search engine. Section 3 presents the results for various settings of our simulation parameters and
gives our explanations for the observed phenomena. In Section 4 we give our conclusions and some ideas for further work.

2. Experimental Design

We have decided to centre our simulations on a simple subset of chess: the KRK endgame. In this endgame White has a King and a Rook, while Black has only a King. The goal for White is to mate the opponent, striving to do so in as little moves as possible. There are two possible outcomes of this endgame: a win for White or a draw. While the KRK endgame is very simple, it still possesses all the interesting attributes: positions are of various difficulties (measured in the number of moves to mate), there surely exist dependencies between the values of sibling nodes in a game tree, and there is a possibility of blunders and early termination for both sides (stalemate or losing a Rook for White; premature mate for Black).

We are interested in the quality of play for White under different conditions. Therefore, unless stated otherwise, we always look at things from the White player’s perspective. Also, White is our MIN player and Black is our MAX player.

For the KRK endgame we have at our disposal an absolutely correct evaluation function. It tells us how many moves are needed to mate in the case that both players play optimally and is measured in moves. It is in the form of a database that consists of all possible legal positions and their evaluations. The database can be obtained from UCI Machine Learning Repository (Blake and Merz, 1998). The positions in the database always assume it is Black’s turn to move. There are two special cases: value 0 means Black is mated and value 255 means that Black has a draw (either the position is a stalemate or Black can capture the white Rook).

The database consists of 28,056 positions. There are actually over 200,000 legal KRK positions, however board symmetries allow for such a reduction. Detailed description of the database and board symmetries is given in Bain (1992). Our version of the database is implemented as an array of 28,056 cells and can be viewed as a sort of transposition table. Apart from positions having special evaluations of 0 or 255, there are 25,233 positions divided into 16 levels of difficulty. Positions from level 1 require one move (2 plies) to mate (assuming optimal play); positions from level 2 require two moves to mate, and so on. Positions from the most difficult level require 16 moves (32 plies) to mate. Different levels have different number of positions; for example, there are 4,553 positions of level 14 and only 390 positions of level 16. Figure 1 shows how many cases (positions) are left unsolved if we applied searches of various depths without any knowledge apart from the rules of the game. The term ‘unsolved’ in this context means
that White has to make a move without knowing at that time the complete move-tree that guarantees a mate. The curve starts to fall significantly between depths of 14 and 20 plies and after ply 20 it steeply drops towards zero.

Figure 1. Number of unsolved cases as a function of search depth.

For the purpose of our experiments we corrupted the ideal evaluation function in a controlled manner. Our method of doing this is as follows. We take a position value and add to it a certain amount of Gaussian noise. The formula and a plot are as follows:

\[ P(x) = e^{-\frac{(x-\mu)^2}{2\sigma^2}} \]

The formula gives the probability \( P(x)dx \) that given the correct evaluation \( \mu \) and standard deviation \( \sigma \) the new (corrupted) evaluation \( x \) will take on a value in the range \([x, x + dx]\), which is a real number. The error of new evaluation is \( \mu - x \). We do this for all positions in the database, including the positions where Black is already mated (special value 0). The corruption is
symmetrical, meaning that there is practically equal chance that the new evaluation will be optimistic or pessimistic. We allow $x$ to take on a negative value – in this way we are able to preserve the symmetry for positions that have true values close or equal to 0.

The level of corruption is controlled by the parameter $\sigma$, which is in fact the standard deviation and which controls how dispersed are the corrupted values $x$ around the correct values $\mu$ (the width of the hill on the plot above). The standard deviation is measured in moves. For example, if $\sigma$ equals 0.5, this means that approximately two thirds of corrupted evaluations are within 0.5 moves around the true evaluation and over 95% of corrupted evaluations are within 1.0 move (two standard deviations) around the true evaluation.

To be able to compare the quality of initial knowledge (evaluation function) to the quality of knowledge after backing up the values with the minimax algorithm, we have to be able to calculate the standard deviation after minimaxing. This is easy, because our search algorithm returns the backed-up values from a fixed search depth for every unique KRK position in an array exactly the same as our initial database. This array is in fact our ‘backed-up’ evaluation function. We thus have one such array for every search depth from 0 (initial database) to 32 ply (our chosen final search depth). After obtaining such an array, we calculate $\sigma$ with the formula:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu_i)^2}$$

where $x_i$ is the backed-up corrupted value and $\mu_i$ is the true value for position $i$. $N$ is the number of positions in the array. This gives us a tool to monitor directly how minimax affects the quality of evaluation function.

Our search engine is the standard fixed-depth minimax search. The only built-in knowledge it has is the ability to detect fatal errors for White (stalemates and losing a Rook); the ability to detect mates is not given. We were able to search to very high search depths of 32 plies and beyond (if desired) by exploiting the fact that the KRK endgame only has a comparatively small number of unique (under symmetries) positions (28,056) which we can all store in a sort of transposition table. We start at depth 0 by loading the values from a (corrupted) database, then move on to depth 2, perform a 2-ply minimax search and use the results of the previous depth as evaluations of the leaves, store results of depth 2 search, move on to depth 4 and so on. Such an implementation of the search algorithm allows it to have a linear time complexity instead of the usual and very constraining exponential time complexity.
3. The Results of the Experiments

Figure 2 shows what happens to the quality of the evaluation function when we change the search depth. The x-axis represents the search depth measured in plies and the y-axis represents the standard deviation $\sigma$ measured in moves. Each curve in the graph represents a different evaluation function – they differ in the level of their corruption (the initial $\sigma$). The legend marks these different evaluation functions with the size of their initial $\sigma$. The best way to separate the curves is to look at their initial corruption. The last evaluation function with initial $\sigma$ of 20 is off the scale and its corruption level never drops. We performed the experiments with several evaluation functions having the same initial $\sigma$, because the introduction of noise is a random process. We found out that the main characteristics remain the same for all evaluation functions with the same $\sigma$ and have therefore plotted just one evaluation function with certain initial $\sigma$ in all the figures.

It seems that we have to divide the evaluation functions into two groups: in the first group we have evaluation functions with a (relatively) low initial error of less than 3.0 and in the second group those with a high initial error. The first group is a realistic model of 'real-life' evaluation functions, while the second group contains evaluation functions with (almost) zero knowledge. We can observe that evaluation functions from the first group do not exhibit the tendency to drop towards 0 (perfect knowledge). They drop slightly or remain on the same level of corruption at best; some even increase. In contrast, evaluation functions from the second group only increase with increased search depth.

The experiment demonstrates that for the evaluation functions tested searching deeper does not improve the quality of the evaluation function for playing the KRK endgame, not even for those with a small initial level of corruption. The endgame undoubtedly contains dependencies between the values of sibling nodes in a game tree. It is also full of possibilities for blunders on the part of white player. White can, after all, lose the Rook in at most two moves if Black plays normally. This means that the two reasons why minimax is believed effective in practice are present and yet the pathology is present as well. How can this be explained?
One thing we were interested in was how the backed-up evaluations are corrupted. We were curious whether backed-up evaluations are excessively optimistic or excessively pessimistic. To this end we calculated the bias of a backed-up evaluation function. Bias is defined as:

\[
bias = \frac{1}{N} \sum_{i=1}^{N} (\mu_i - x_i)
\]

where \( \mu_i \) and \( x_i \) are again the true and backed-up value of position \( i \), respectively. If bias is highly negative then the backed-up values are generally overly pessimistic and if bias is highly positive then the backed-up values are generally overly optimistic. Since the noise introduced into the various evaluation functions was symmetrical we expected the bias to be close to zero, meaning some backed-up evaluations are too optimistic and others too pessimistic. Figure 3 charts how biased various evaluation functions are with respect to search depth. All curves start in close proximity of zero and then without exception they all exhibit a highly positive bias. The higher the level of initial corruption the higher the bias gets. Most of the bias is acquired in transition from search depth 0 to search depth 2. These two levels differ the most of any two consecutive levels – depth 0 means the algorithm goes directly to the lookup table, while on level 2 it performs minimaxing for the first time. Other transitions just increase the depth of minimaxing.
If we look closely at what is happening at the last level of minimaxing we can come up with an explanation why the bias occurs. On the last level we either have a max or a min operation. In our case we always had White to choose on the last level which meant a min operation. In presence of noise the operation of choosing a minimum value will be biased towards lower values. This is not saying that the value chosen will always be lower than it would be without noise, but in general this will be so much more often than not. We can thus see that the lowermost operation of the minimax algorithm introduces a bias. But surely the opposite operation, finding a maximum, which follows on the next higher level should (partly) negate this bias? It does not, however, because the majority of the values it operates on are already biased and all it does is selecting one of them. The bias is actually introduced on that lowest level of minimaxing.

If we look at Figures 2 and 3 simultaneously we find out that the corruption level and the level of bias are highly correlated. For evaluation functions with a lower initial level of corruption (0.25 and 0.50) this correlation begins to manifest with the higher search depths of 16 to 20, while for others it begins much sooner, from a search depth of 10 for curve 0.75 and from a search depth of 4 for curve 1.0. For curves with initial corruption higher than 1.0 the correlation is strong immediately from search depth 2 onward. It is no wonder then that backed-up evaluation functions could not get any better – they were prevented from doing so by the bias. However, bias, at least in general, equally affects all evaluations. It
Search versus Knowledge: An Empirical Study of Minimax on KRK

resembles adding a constant to all evaluations. This in turn means that we do not change the order of the available moves relatively to one another in the position we are trying to evaluate. If this is true, then minimax actually does improve the evaluation, but on the surface it is not seen, because of the bias.

Figure 4. Influence of search depth on quality of evaluation function with bias accounted for.

To confirm this claim, we calculated another statistic, a standard deviation as before. However, this time we have taken into account that all the values were shifted away from the true values by the bias. The formula for this statistic, \( \sigma' \), is:

\[
\sigma' = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - (\mu_i + \text{bias}))^2}
\]

where \( \mu_i \) and \( x_i \) are again the true and backed-up value of position \( i \), respectively. How this new standard deviation changes in relation with search depth is shown in Figure 4. Here we can see that it drastically falls with the increase of search depth. Again, the evaluation functions from the two groups defined earlier behave differently. Evaluation functions from group one drop and then stay more or less on the same level, while evaluation functions from group two first drop and then start to rise back up again. This positive effect of the minimax principle with increasing search depth on the evaluation functions from group one is exactly the result that
Beal was expecting from his model in 1980, but he was unable to prove it because the model did not account for the introduced bias.

Up to this point, we did not say anything about how well a computer program using one of our corrupted evaluation functions would actually play. The answer is given in Figure 5. We have played out all unique KRK positions except the ones with special values of 0 or 255, in total 25,233 positions. White was guided by a corrupted evaluation function. Additionally, White was allowed to use a simple mechanism to avoid repeating the same position over and over again. The mechanism kept a list of all positions that already occurred in the game and if the position was to be repeated a different move was selected (the next best move according to the evaluation function). Black was always playing optimally. We measured the quality of play as the average number of moves above what an optimal white player (using a non-corrupted database) would need. This statistic is computed as the difference between the number of moves spent by White for all positions and the number of moves needed for all positions using optimal play, divided by the number of positions (25,233). The curves representing play using evaluation functions with initial corruption level of 5 and 20 are off the scale and result in play that is not even able to mate the opponent within the required 50 moves.

In Figure 5 we can see that an evaluation function with initial corruption level of 0.25 moves provides practically optimal play starting already at search depth 0. The quality of play using other evaluation functions gradually increases with deeper searches until it reaches a sort of threshold for a given evaluation function. From that point onward the quality of play remains more or less on the same level. This is true for evaluation functions with initial corruption level below 3. Those evaluation functions with initial corruption level higher than 3 result in a play that is not even good enough to mate the opponent within the required 50 moves. We can observe a correlation between the quality of play in Figure 5 and the knowledge corruption level of the evaluation functions in Figure 4.
4. Conclusions and Further Work

Some theoretical studies of the minimax principle in the past have shown that it has a negative effect on the quality of the evaluation function. As the answer why it is nevertheless successful in practice they suggested two reasons: (a) dependencies between the true values of sibling nodes in a game tree, and (b) existence of traps that cause early terminations of the game.

We have taken the opposite approach to the problem; we tried to check empirically these conclusions using the KRK chess endgame. We can confirm that the minimax algorithm appears to be a poor preserver of the knowledge built into the evaluation. Yet, regardless of that, it proved to be still successful in actual play (Figure 5). It turns out that even with dependencies between evaluations of sibling nodes in a game tree and an abundance of possibilities to commit blunders present in our endgame, the anomaly still existed.

However, we claim that the minimax principle in combination with noisy evaluation functions introduces a bias into backed-up evaluations. This bias is the culprit why mathematical models could not prove the effectiveness of the minimax that was observed in practice. Once bias is properly accounted for, the positive influence of the minimax principle with increasing search depth is unmasked. The main problem is that bias moves the backed-up evaluations away from the true values, hence causing the illusion that they

![Image: Noise effect on actual play](image_url)

**Figure 5.** Quality of play using corrupted evaluation functions.
are more corrupted. Yet, since it more or less affects all the evaluations equally it does not affect the relative ordering of the available moves with respect to their quality. So, if we look at the evaluations in absolute terms they are increasingly corrupted with a growing bias, but if we look at them in relative terms they become progressively better with higher search depths.

In view of the presented results it would be very interesting to recheck our results using a more complex game – perhaps a KQKR or KRKN chess endgame, or some artificially designed game. A further study of how the bias behaves and what affects it is also necessary. Especially interesting would be to investigate what happens with the bias if we mix the functions (min and max) at the lowest level of minimaxing – some branches we search to even depths, others to odd depths.

References


STATIC RECOGNITION OF POTENTIAL WINS IN KNNKB AND KNNKN

E.A. Heinz*
International University (IU), School of IT, Kasernenstr. 12, D-76648 Bruchsal, Germany
ernst.a.heinz@web.de; http://www.i-u.de/schools/heinz/

Abstract  The fact that the strong side cannot enforce a win in KNNK makes many chess players (both humans and computers) prematurely regard KNNKB and KNNKN to be trivially drawn too. This is not true, however, because there are several tricky mate themes in KNNKB and KNNKN which occur more frequently and require more complicated handling than common wisdom thinks. The text analyzes the mate themes and derives rules from them which allow for the static recognition of potential wins in KNNKB and KNNKN without further lookahead by search. Although endgame databases achieve the same goal, they are normally far less efficient at doing so because of their additional I/O and memory requirements (even when compressed).

Keywords: computer chess, endgame play, KNNKB, KNNKN, static recognition

1. Introduction

Usually, two bare Knights are not much of a force when it comes to mating in late endgames such as KNNK, KNNKB, and KNNKN. It is well-known that these endgames are generally drawn despite the substantial material advantage enjoyed by the strong side (Thompson, 1991; The Editors, 1992; Nalimov, Haworth, and Heinz, 2000, 2001). Human chess players and chess programs alike tend to incorporate rules of thumb classifying bare KN constellations as most unlikely to win. Thus, common chess wisdom avoids KNN types of positions when being ahead in material and goes for them otherwise. A crude way to implement the heuristic is by scoring essentially all KNNK, KNNKB, and KNNKN positions as draws. Like many others, an early version of our own chess program DARKTHOUGHT (Heinz, 1997, 2000) did exactly this back in

*This work originally started back in the mid-1990s while the author still was a Ph.D. candidate at the School of Computer Science, University of Karlsruhe, Germany, and then continued throughout his stay as a postdoctoral fellow at the M.I.T. Laboratory for Computer Science, USA, from 1999 to 2001.
mid-1995. Then, at the end of some blitz test games, it encountered the two positions shown in Figures 1 and 2 where it happily went for the continuations leading to Figures 3 and 4 respectively, mistakenly scoring them both as draws. DARKTHOUGHT played without endgame databases and short on time, so it saw the loss only after having manoeuvered itself into it.

Of course, the aforementioned scenario with the so-called "horizon effect" visible at low search depths is nothing unusual in computer chess. It was quite special, however, that the horizon effect occurred with full severity (score drop-
ping from draw to being mated) in seemingly trivial circumstances (KNNKB and KNNKN). This strongly aroused my curiosity and sparked the work that eventually led to the development of interior-node recognizers (Heinz, 1998, 2000), knowledgeable RAM-based endgame databases (Heinz, 1999a, 2000), and efficient endgame indexing (Heinz, 1999b, 2000; Nalimov et al., 2000, 2001) plus their implementation in DARKTHOUGHT. Hence, those rather innocent-looking two positions from Figures 1 and 2 were in fact instrumental for much of my endgame-related research up to date.

Solving the KNNKN "mate in 3" of Figure 4 requires a 5-ply search with 4 quiet half-moves before the final checkmate: White’s 1. Nc3 {Nf4} and 2. Nd5 plus Black’s respective answers. Consequently, standard quiescence searches following either checks and captures or captures only cannot spot the win unless supported by lucky hits in the transposition table. The same holds for normal full searches with remaining depths of ≤ 3 plies in case of capture-check quiescence and ≤ 4 plies in case of capture-only quiescence. Therefore, the search alone most likely fails to resolve the mate in this simple position if it occurs far out near the lookahead boundary. According to Thompson (1991), The Editors (1992), and Nalimov et al. (2000, 2001) the endgames KNNKB and KNNKN feature even harder positions than the ones from Figures 3 and 4: the longest forced win for KNNKB is "mate in 4" (see Figure 5) and for KNNKN it is "mate in 7" (see Figure 6). Because of the checks and single-reply moves involved here, normal full searches with extensions and quiescence searches with checks included might actually resolve these deeper mates more easily than my two example positions with their many quiet moves.

![Figure 5](image1.png)  **Figure 5.** White mates in 4 moves: 1. Nb6+ Kb8 2. Nd7+ Ka8 3. Kc7 Bh1 {or any other legal move by B} 4. Nb6#.

![Figure 6](image2.png)  **Figure 6.** White mates in 7 moves: 1. Na6+ Kb7 2. Ne5+ Kb8 3. Ne7 Ng3 4. Nc6+ Ka8, 5. Kc7, 6. Nd7, 7. Nb6#.
An obvious solution to the problem is the usage of omniscient endgame databases that return the exact distance-to-win (mate or conversion to another won subgame) when queried. In practice, this does not really work out because endgame databases (even in compressed format) usually reside on secondary storage media due to their large sizes. Thus, their querying incurs considerable performance penalties and additional memory consumption for caching purposes. As a good compromise between accuracy and speed, most chess programs do not query any endgame databases in the quiescence search and very often stop doing so a few plies above the main lookahead boundary already. Please note, however, that in the particular case of KNNKB and KNNKN it is possible to copy Nalimov's compressed tablebases (Nalimov et al., 2000, 2001) to a RAM disk requiring less than 1 MB of memory and access them from there. Performance-wise, the necessary I/O, index calculations, and data decompression still lose against the static recognition rules suggested by me later on in this text – but actually not by much. Yet, the special database setup does not allow for any generalization regarding other similar positions. In particular towards this end, I see excellent promise of the rule-based approach though.

The remainder of this text is structured as follows. The next section discusses related work. Then, the subsequent sections focus on the various mate themes in KNNKB, KNNKN, and their subgames (namely KBKN, KNKN, and KNNK). These themes lead to the derivation of recognition rules and the final formulation of the full algorithm for the static recognition of potential wins in KNNKB and KNNKN. Last but not least, a wrap-up of the main findings and a look into the future conclude the work.

2. Related Work

There exists an ample body of related work covering endgame databases and infallible rule-based endgame play in chess. Both areas feature a long and rich history of interesting research. The introductory section above already referred to some important contributions in the field of endgame databases, namely (Thompson, 1991; The Editors, 1992; Nalimov et al., 2000, 2001). An elaborate discussion of endgame databases and their history was provided by Heinz (1999b, 2000). The introduction also mentioned the interior-node recognizers and knowledgeable endgame databases of DARKTHOUGHT (Heinz, 1998, 1999a, 2000). Both are of special interest here because the static recognition rules for KNNKB and KNNKN are to augment that very recognizer framework.

The rest of this section now focusses on infallible rule-based endgame play in chess. As early as 1890, Torres y Quevedo built a marvelous electro-mechanical machine which played and won many of the hardest KRK positions. Tan (1972) implemented the first program that achieved seemingly infallible play for the KPK endgame. Tan's excellent set of rules solved all difficult KPK positions
known by then, including Averbakh’s and Fine’s famous examples. But because omniscient KPK endgame databases were not yet available in 1972, the hypothesized perfectness of the program remained unproven. However, Tan’s program doubtlessly pioneered the usage of decision trees with multi-valued nodes and leaves representing specific pattern knowledge about the respective endgame domain. Decision trees became an integral part of nearly all subsequent works which focussed on the explicit construction or automatic deduction of complete rule sets for infallible endgame play in chess. Later on, Bratko, Kopec, and Michie (1978), Bramer and Clarke (1979), and Bratko and Michie (1980) presented more refined representation schemes for pattern knowledge in chess endgames. Several good examples of these and other predominantly hand-crafted rule sets are listed below.

- **KBNK** – van den Herik (1983);
- **KNNKP(h)** – Herschberg, van den Herik, and Schoo (1989);
- **KNP(h)K** – van den Herik (1980, 1982);
- **KPK** – Tan (1972), Beal (1977), Beal and Clarke (1980), Bramer (1980a, 1980b), Niblett (1982), Barth and Barth (1992);
- **KPKP** (both P passed) – Bratko (1982), Barth (1995);
- **KRK** – Torres y Quevedo [1890], Zuidema (1974), Bramer (1980a, 1982);
- **KRKN** – Bratko and Niblett (1979), Kopec and Niblett (1980);
- **KQKP** – Barth and Barth (1992);
- **KQKQ** – Barth and Barth (1992), Weill (1994).

The surprising complexity of rules and knowledge bases for “simple” problem domains (such as 3-piece endgame databases in chess) ignited the interest of researchers in learning such infallible rules automatically. Especially the KRK and KPK endgame databases became extremely popular for automatic learning experiments. Many works that try to automate the inductive acquisition of rules for infallible endgame play in chess also employ decision trees as their central resources for the representation of semantic knowledge. The following brief overview of publications about automatic learning of infallible endgame play in chess and the inductive acquisition of rule-based knowledge therefor is meant to serve as a mere introduction to the field. Any more comprehensive summary clearly lies beyond the scope of this text.

- **KBBKN** (selected positions) – Muggleton (1988);
- **KBRK** (extended chess version) – Coplan (1998);
- **KNNKP(h)** – van Tiggelen and van den Herik (1991), van Tiggelen (1991, 1998);
- **KPK** – Michalski and Negri (1977), Negri (1977), Shapiro and Niblett (1982), Shapiro (1987), Coplan (1998);
- **KP(a7)KR** – Shapiro and Michie (1986), Shapiro (1987), Muggleton (1990);
- **KRK** – Bain (1994), Bain and Muggleton (1994), Bain and Srinivasan (1995);

### 3. Checkmates in KBKN and KNKN

Although the subgames KBKN of KNNKB and KNKN of KNNKN are trivially drawn, they actually do feature a few checkmate positions. These are shown in Figure 7 where each quadrant of the board depicts its own mate theme to be viewed independently of the others. Yet, normal non-mate positions in KNKB and KNKN do never forcibly lead to the side on move being mated (i.e., there are only direct “mates in 1” because the side on move can always evade any mating attempt). Still, the mate themes of Figure 7 demand proper attention because they also apply to KNNKB and KNNKN where both the strong as well as the weak side might mate the opponent accordingly. The static recognition rules must take all these possibilities into full account.

![Figure 7. Mate themes in KBKN and KNKN (corner traps, not enforceable).](image-url)
4. Checkmates in KNNK

Despite the considerable material advantage of the strong side, the subgame KNNK of KNNKB and KNNKN is generally drawn too (Thompson, 1991; The Editors, 1992; Nalimov et al., 2000, 2001). Again, there are no enforceable checkmates in KNNK but the number and variety of mating themes and direct mates are much higher here than in the materially balanced subgames KBKN and KNKN. Figure 8 visualizes whole sets of mate themes in KNNK by showing several alternative locations of the strong King together with a fixed
placement of the two Knights and the weak King in the same single quadrant. The checkmate positions in each such set differ solely by the location of the strong King. Like the mate themes of KBKN and KNKN, these KNNK checkmates all involve trapping the weak King in a corner of the board. In addition to the corner traps, there is another special mate theme in KNNK that works by trapping the weak King on the edge of the board away from the corner. Figure 9 depicts this additional mate theme for one possible location of the weak King on the edge in the upper half of the board. The theme remains valid when shifting it to the left or right within the bounds of the board, of course. Although not being actively enforceable within KNNK, all the mate themes of Figures 8 and 9 exemplify potential wins by the strong sides in KNNKB and KNNKN. Therefore, the static recognition rules must also cover them properly.

5. Checkmates in KNNKB and KNNKN

Those readers who are still not convinced that KNNKB and KNNKN positions deserve better than being scored as some kind of draw might finally reconsider after taking a look at the following numbers found in Nalimov's tablebase summary files (Nalimov et al., 2000, 2001). Roughly 10,000 position templates in KNNKB and 40,000 position templates in KNNKN are won for the KNN side. The vast majority of them are non-direct forced wins requiring several moves to mate. Including symmetries, the real numbers of won positions for the KNN side amount to $4x - 8x$ as many: i.e., 40,000 to 80,000 in KNNKB and 160,000 to 320,000 in KNNKN.

Compared with the 300 to 600 forced wins of KNNK (all direct mates), there are orders of magnitude more forced wins in KNNKB and KNNKN where the weak side features a minor piece in addition to the King. Hence, the KNN side is actually better at enforcing checkmate if the opponent defends itself with more material than just a lone King. As counter-intuitive as this might seem at first glance, it is quite well-known and not too hard to understand because the additional piece prevents stalemates and may even block an escape route of the weak King. However, the added material also enables the weak side to mate the opponent in some non-enforceable circumstances brought about by bad play of the strong side. Consequently, KNNKB and KNNKN contain some positions where the weak side wins and the KNN side is mated.

5.1 Weak Side Wins

The mate themes of all subgames still apply in KNNKB and KNNKN as well, with the excess piece (a strong Knight) located anywhere else on the board. The famous forced wins in $\leq 7$ moves with a single Knight against a Pawn in KNKP(a,h) exploit the very same strategy.
board in legal fashion. If the second Knight of the KNN side also resides directly beside the strong King trapped in a corner, the noteworthy additional mate themes shown in Figure 10 arise. The static recognition rules must take all such possibilities into full account.

5.2 KNN Side Wins

As before, the mate themes of all subgames apply in KNNKB and KNNKN too. On top of these, the KNN side may now mate the opponent even without
any support of its own King. Figure 11 presents the according NN-checkmates which are quite exceptional and not enforceable. Other additional mate themes of KNNKB involving the full set of 5 pieces on the board are shown in Figures 12 and 13 (corner traps) and Figure 14 (edge traps). This overview of positions with the KNN side winning is by no means exhaustive. But due to space limitations, the remaining positions won by the strong side in KNNKB cannot be shown here. Unfortunately, the very same holds for all additional KNNKN mate themes and positions won by the strong side there. Nevertheless, the static recognition rules must of course cover them all in a suitable way too.

![Figure 12](image12.png)  
**Figure 12.** Additional mate themes for strong side in KNNKB (I).

![Figure 13](image13.png)  
**Figure 13.** Additional mate themes for strong side in KNNKB (II).
6. Static Recognition Rules

The preceding sections on checkmates in KNNKB and KNNKN plus all their subgames (KBKN, KNKN, KNNK) argue that all possible mate themes in these endgames involve trapping the enemy King in either the corner or on the edge of the board. The omniscient endgame databases confirm this notion but their exhaustive querying also reveals some forced wins for the KNN side in KNNKN where the weak King resides on one of the "extended corner" squares of the board, namely b2, b7, g2, and g7. Figure 15 shows such a position which arises from the forced win in 7 moves of Figure 6 after 1. Na6+ Kb7.

Figure 14. Additional mate themes for strong side in KNNKB (III).

Figure 15. White mates in 6 moves – see Figure 6 after 1. Na6+ Kb7.
The ensuing motif how to enforce the final checkmate does not work against a defending Bishop. Hence, there are no forced wins with the weak King located on "extended corner" squares in KNNKB.

**Strong-Win Potential.** The winning chances of the strong crucially hinge on its ability to keep the weak King trapped on the edge and, in case of KNNKN, the "extended corner squares of the board. Success in doing so is quite tedious to determine exactly because of possible checks, attacks on the strong Knights, and even pins by the weak Bishop in KNNKB.

**NN-Mate Rule.** If the weak King is located in a corner of the board with its Bishop or Knight directly beside it on an "extended corner" square and a strong Knight trapping it from the next square on the long diagonal, then the special mate themes of Figure 11 loom. They do not require any direct support by the strong King. So, the position is a guaranteed win for the KNN side if the other strong Knight already gives a check or is on move and able to deliver a direct check (in KNNKN this holds even if the strong King is currently in check itself). Otherwise, the position is drawn in KNNKB if the weak side is on move or the strong side is in check because then the weak Bishop can capture a Knight (see Figure 11).

**Weak-Draw Rule.** If the weak King does not reside on the edge of the board and not on any "extended corner" square in case of KNNKN either, then the weak side at least draws. The same holds if the weak side is on move and the weak King can directly step off the edge and the "extended corner" in case of KNNKN. If the distance between the two Kings exceeds 4 steps measured in squares on the board, the position is drawn too as discovered by exhaustive analyses of the endgame databases KNNKB and KNNKN. Depending on the side-to-move and whether it is a KNNKB or KNNKN position, the distances between the two Kings triggering a draw are even smaller (see recognition algorithm below for more details).

**Weak-Win Rule.** If the strong King is located in a corner of the board with at least one of its Knights directly beside it on the edge of the board and the weak King covers the "extended corner" square next to the strong King, then the weak side might even win whereas the strong side at most draws. If so and the strong side is on move but not checkmated, then the position is drawn. If so and the weak side is on move but cannot directly check and mate the opponent, then the position is drawn as well.
7. Static Recognition Algorithm

Constant and Type Declarations

```plaintext
TYPE boardstate = ...; /* state of a given position on chess board */
TYPE score = ...; /* range of valid scores */
TYPE side = ENUM {black, white};
TYPE square = ENUM {al, ..., hl, ..., a8, ..., h8};

const SET OF square: corner = {a1, h1, a8, h8};
const SET OF square: edge = {a1, ..., h1, a2, h2, ..., a7, h7, a8, ..., h8};
const SET OF square: xcorner = {b2, g2, b7, g7};
```

**KNNK[B,N] Recognition Function**

```plaintext
FUNC score knn_k_b_n_recog(const boardstate: pos; const side: strong, weak) {
  const square: strong_k = k_sqr(strong, pos);
  const SET OF square: strong_k_area = k_attck(strong_k);
  const SET OF square: strong_nn = n_sqrs(strong, pos);
  const SET OF square: weak_b_n = b_sqrs(weak, pos) + n_sqrs(weak, pos);
  const square: weak_k = k_sqr(weak, pos);
  const SET OF square: weak_k_area = k_attck(weak_k);
  const square: weak_minor = ANYELEM(weak_b_n);

  IF (strong_k IN corner) /* strong K trapped by own Ns and weak K with */
    & & EMPTY(strong_k_area - strong_nn - weak_k_area) /* no escape */
  {
    const SET OF square: b_mates = xcorner * strong_k_area; /* target */
    const SET OF square: n_mates = n_attck(strong_K); /* squares for */
    /* B, N to mate strong K */
    IF (side_to_move(pos) == strong) /* weak side on move may mate */
      & & ((is_knnkb(pos) & & !EMPTY(weak_b_n * b_mates))
        || (is_knnkn(pos) & & !EMPTY(weak_b_n * n_mates)))
      RETURN stm_mated_score(pos)
    ELSE IF (side_to_move(pos) == weak) /* otherwise, position is drawn */
      & & EMPTY(n_attck(weak_k) * strong_nn) /* if not in check */
      & & ((is_knnkb(pos) & & !EMPTY(b_attck(weak_minor, pos) * b_mates))
        || (is_knnkn(pos) & & !EMPTY(n_attck(weak_minor) * n_mates)))
      RETURN stm_mates_score(pos);
    ELSE /* drawn if weak K not on edge */
      RETURN draw_score(pos);
  }

  ELSE /* otherwise, position is drawn */

  IF (weak_k IN corner) /* weak K in corner trapped by own B, N on */
    & & !EMPTY(weak_k_area * xcorner * weak_b_n) /* "extended corner" */
```

---

<table>
<thead>
<tr>
<th><strong>Static Recognition of Potential Wins in KNNKB and KNNKN</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>57</td>
</tr>
</tbody>
</table>

---

**Constant and Type Declarations**

- `TYPE boardstate = ...; /* state of a given position on chess board */`
- `TYPE score = ...; /* range of valid scores */`
- `TYPE side = ENUM {black, white};`
- `TYPE square = ENUM {al, ..., hl, ..., a8, ..., h8};`

**KNNK[B,N] Recognition Function**

```plaintext
FUNC score knn_k_b_n_recog(const boardstate: pos; const side: strong, weak) {
  const square: strong_k = k_sqr(strong, pos);
  const SET OF square: strong_k_area = k_attck(strong_k);
  const SET OF square: strong_nn = n_sqrs(strong, pos);
  const SET OF square: weak_b_n = b_sqrs(weak, pos) + n_sqrs(weak, pos);
  const square: weak_k = k_sqr(weak, pos);
  const SET OF square: weak_k_area = k_attck(weak_k);
  const square: weak_minor = ANYELEM(weak_b_n);

  IF (strong_k IN corner) /* strong K trapped by own Ns and weak K with */
    & & EMPTY(strong_k_area - strong_nn - weak_k_area) /* no escape */
  {
    const SET OF square: b_mates = xcorner * strong_k_area; /* target */
    const SET OF square: n_mates = n_attck(strong_K); /* squares for */
    /* B, N to mate strong K */
    IF (side_to_move(pos) == strong) /* weak side on move may mate */
      & & ((is_knnkb(pos) & & !EMPTY(weak_b_n * b_mates))
        || (is_knnkn(pos) & & !EMPTY(weak_b_n * n_mates)))
      RETURN stm_mated_score(pos)
    ELSE IF (side_to_move(pos) == weak) /* otherwise, position is drawn */
      & & EMPTY(n_attck(weak_k) * strong_nn) /* if not in check */
      & & ((is_knnkb(pos) & & !EMPTY(b_attck(weak_minor, pos) * b_mates))
        || (is_knnkn(pos) & & !EMPTY(n_attck(weak_minor) * n_mates)))
      RETURN stm_mates_score(pos);
    ELSE /* drawn if weak K not on edge */
      RETURN draw_score(pos);
  }

  ELSE /* otherwise, position is drawn */

  IF (weak_k IN corner) /* weak K in corner trapped by own B, N on */
    & & !EMPTY(weak_k_area * xcorner * weak_b_n) /* "extended corner" */
```
58

E.A. Heinz

```
& & !EMPTY(strong_nn * {c3, f3, c6, f6} * k_attck(weak_minor))
{ /* and by strong N in diagonal opposition */
  IF !EMPTY(strong_nn * n_attck(weak_k)) /* weak K also in check by */
    RETURN stm_mated_score(pos); /* 2nd strong N => checkmate! */

  IF is_knnkb(pos) && ((side_to_move(pos) == weak) || (strong_k IN
    (k_attck(weak_minor) * {c1, f1, a3, h3, a6, h6, c8, f8})))
    RETURN draw_score(pos); /* drawn in KNNKB if weak side on move */

  RETURN (is_knnkb(pos) || (is_knnkn(pos) ? xcorner)) /* strong side on move and other strong N ready to deliver mate */
  IF (side_to_move(pos) == strong) && !EMPTY(n_attck(weak_k) * n_attck(
    ANYELEM(strong_nn - {c3, f3, c6, f6} * k_attck(weak_minor))))
    RETURN (is_knnkb(pos) || (strong_k IN n_attck(weak_minor)))
    ? stm_mates_score(pos) : stm_mates_next_score(pos);

  RETURN rcg_fail_score(pos); /* weak side on move in KNNKN ==> */
  /* may still draw (unwind the trap by removal of N) */
```

/***** WEAK-DRAW PART (II) *****
/* drawn if K distance > 4 steps */
```
IF sqr_dist(strong_k, weak_k) > 4 RETURN draw_score(pos);
```

RETURN rcg_fail_score(pos); /* handle tricky issues by further search */
/* and trigger an extension in this line */
```

7.1 Algorithm Description

**Auxiliary Functions.** The recognition algorithm relies on several auxiliary functions not specified in detail here. There are a number of routines to access and query the current state of the chess board passed in the parameter pos of type boardstate: k_sqr returns the King location of the desired side; b_sqrs and n_sqrs return the locations of all Bishops and Knights respectively for the desired side; is_knnkn and is_knnkb identify the exact material balance; and side_to_move returns the side on move in the given position. Another group of auxiliary functions handles the encoding of recognizer failures, checkmates, draws, and mates in this or the next move after it into valid scores: rcg_fail_score, stm_mated_score, draw_score, stm_mates_score, and stm_mates_next_score. The numerical function sqr_dist returns the distance between two squares on the board as measured in single-square steps that a King needs in
moves on an empty board to travel from one to the other. Last but not least, the algorithm requires support for the calculation of sets of squares attacked by Bishops, Kings, and Knights located anywhere on the board. The functions b_attck, k_attck, and n_attck perform the according attack generations for B, K, and N respectively. The sliding coverage of Bishops along their diagonals specifically depends on the full board state, whereas Kings and Knights always attack the same sets of squares from a given location regardless of any other pieces.

Constants and Types. The constant sets corner, edge, and xcorner capture the important corner, edge, and "extended corner" squares of the chess board. The enumeration type side contains just two items: black and white. The enumeration type square covers all board squares denoted by the 64 items a1, ..., h1, ... a8, ..., h8. The anonymous types boardstate and score represent the full states of chess positions and scoring values respectively.

Pattern Recognition. The algorithm applies basic set operations on sets of squares to achieve location-independent pattern recognition. As an example take the core NN-mate pattern of the weak King in any corner, the weak Bishop or Knight directly beside it on the corresponding "extended corner" square, and one of the strong two Knights diagonally beside the weak minor piece as depicted in Figure 11. The membership test weak...k_IN corner assures that the weak King resides in a corner. Then, the intersection weak...k_area * xcorner * weak_b_n gives the set of "extended corner" squares with a weak Bishop or Knight directly beside the weak King. If the set is not empty, it contains the square of the weak minor piece as a single element and the second pattern condition holds. Finally, intersecting k_attck(weak minor) * {c3, f3, c6, f6} * strong.nn computes the set of squares with strong Knights directly and inwardly beside the weak minor. If this set is not empty, the full core pattern is identified independent of the specific corner square the weak King is located on.

Weak-Win Part. The recognition starts with the exceptional wins by the weak side where the strong King is trapped in a corner by at least one of its Knights and the weak King. Depending on which side is on move and whether the weak minor piece can actually deliver a checkmate, the algorithm returns mate or mated scores and a draw score otherwise. A clever trick used here to determine if a single square is attacked by any piece from a set of like pieces works as follows: call the specific attack function of the given piece type with the very square in question as the location parameter, then intersect the resulting attack squares with the original set of like pieces → if and only if the intersection is not empty, the square
in question is under attack by some piece from the set of like ones. This scheme excels at check detection. The term `\text{EMPTY}(\text{attack}(\text{weak.k}) \ast \text{strong.nn})`, for instance, assures that the weak King is not in check by any of the strong Knights.

**Weak-Draw Part (I).** This straightforward section detects draws by the rule that the weak King is not on the edge of the board and not on any “extended corner” squares in KNNKN either.

**NN-Mate Part.** The paragraph on pattern recognition above already discussed the core NN-mate pattern and its recognition in detail. After establishing that the core NN-mate pattern applies, the algorithm tests for checkmate by the second strong Knight attacking the weak King, for draws in KNNKB with the weak side on move or the strong side in check, and for forced mates by the strong side with the second strong Knight ready to deliver the final check. Otherwise, the weak side is on move in KNNKN and may still draw by removing the weak Knight from the “extended corner” square, thus unwinding the trap. The static recognizer intentionally fails at this point in order to resolve the resulting complications of checks and Knight forks by further search.

**Weak-Draw Part (II).** First, the algorithm detects draws by the rule “Kings more than 4 steps apart”. Then, the next draw detection deals with the case that the weak side is on move and may directly step off the edge and the “extended corner” in case of KNNKN. The available escape squares of the weak King are those squares around it not blocked by the weak minor piece and not attacked by either the strong King or its Knights. If the set difference of these escape squares and the edge of the board (plus the “extended corner” squares in case of KNNKN) is not empty, then the weak King directly escapes from the trap and the position is drawn.

**Strong-Win Part.** Whenever no obvious drawing rule for the weak side applies, the static recognizer fails. In case of KNNKN, the weak King still seems to be trapped on the edge of the board or the “extended corner” squares. Further search then resolves the tricky issues of possible checks, attacks on the strong Knights, and pins of the weak Bishop in KNNKB. In general, such explicitly intended failures of static recognizers should trigger search extensions in the current line. If so desired, more ambitious analyses of the piece constellation and attack relations on the board aiming for an even better identification of real wins in KNNKB and KNNKN may easily be added in front of the fail-value return at the end.
7.2 Algorithmic Complexity

The recognition algorithm heavily depends on sets of squares and basic operations on them: set difference, element count, emptiness, intersection, membership, member selection, and union. Other important auxiliary functions are those for attack generation and access to the data structure holding the full state of the current board position.

Sets of Squares. There are 64 squares on a chess board. Hence, the best way to handle sets of squares is by means of a standard bit-vector representation with exactly 64 bits (one for each square) where square \( i \) is in the set if and only if the \( i \)-th bit of the vector is 1. Thus, sets of squares nicely map to 64-bit unsigned integers which are natural data types of modern CPUs. In computer chess such 64-bit values are also known as “bitboards”.

Basic Set Operations. For sets represented as bit vectors, all basic set operations map to simple constant-time computations involving unsigned 64-bit data: \( \text{difference} \to \text{bit-wise AND complement}, \text{element count} \to \text{count bits} \) (a.k.a. population count), \( \text{emptiness} \to \text{compare with 0}, \text{intersection} \to \text{bit-wise AND}, \text{membership} \to \text{test bit}, \text{selection} \to \text{find bit}, \text{and union} \to \text{bit-wise OR} \). Most of these computations actually finish within a single clock cycle on modern CPUs. The 64-bit unsigned integer value 0 represents the empty set and comparisons for set equality are done by standard tests comparing 64-bit unsigned integer values.

Attack Generation. The squares attacked by Kings and Knights depend on their specific locations only, regardless of the placement of any other pieces. Straightforward table lookups indexed by square numbers suffice to perform the according attack calculations \( k\text{-attck} \) and \( n\text{-attck} \). Bishops, on the other hand, are sliding pieces that depend on the full board constellation to determine the exact extent of their attack coverage. Even if implemented by looping over squares in the four diagonal directions, the respective attack calculations of \( b\text{-attck} \) are constant-time bound because their are at most 13 squares to traverse (7 on the middle diagonal of the board and another 6 on one next to the middle). Moreover, so-called “rotated bitboards” (Hyatt, 1999; Heinz, 1997, 2000) enable the full Bishop attack calculations to be done by a few table lookups.

Remaining Auxiliary Functions. Except for attack generation, the auxiliary functions either encapsulate simple access protocols to the data structure carrying the current state of the chess board or they perform equally simple score value encodings. All these computations are constant-time bound and take only a few clock cycles to finish on modern CPUs. The same holds for \( \text{sqr\_dist} \), an auxiliary function not covered up to now: 

\[
\text{sqr\_dist}(x,y) = \max(\ \text{ABS}(\text{VAL}(x)/8 - \text{VAL}(y)/8), \ \text{ABS}(\text{VAL}(x)\%8 - \text{VAL}(y)\%8) )
\]
All in all, the recognition algorithm contains only constant-time bound computations and no loops. Hence, it is of constant time complexity in $O(1)$. As the average and longest execution paths through the algorithm are short and most of the calculations actually finish within a few clock cycles on modern CPUs, the whole algorithm also features good efficiency in practice where acceptably small constants cap its average and worst-case execution times.

8. Conclusion and Future Work

Hundreds of thousands of positions in KNNKB and KNNKN are won for the KNN side. Tricky mate themes occur more frequently and require more complicated handling in these two endgames than common wisdom makes people think. In fact, they are not trivial at all! This paper may very well be the first ever to present a rule-based static recognition algorithm for any complete non-trivial 5-piece endgame because the fine works by Herschberg et al. (1989), van Tiggelen and van den Herik (1991), and van Tiggelen (1991, 1998) consider only the subset KNNKP(h) of the full KNNKP endgame.

All mate themes and rules were developed a-priori by hand. Then, later on, their validity was checked against omniscient endgame databases a-posteriori. In particular, the “trapped King” feature seems very important and powerful for endgames in general and is probably good for static recognition in other endgames as well. Such trapping and the number of escape squares for each King could possibly be used as a crucial position feature and input parameter for machine-learning algorithms that try to extract useful knowledge from endgame databases automatically. The trap patterns look interesting for chess problem composers, too, who have certainly discovered them on their own already.

In the future, I like to use the KNNKB and KNNKN recognition rules as a foundation to statically detect possible draws and “mates in X” in other positions not covered by endgame databases directly (e.g., additional material might not save Black in Figure 3). Moreover, one can still extend the current algorithm to include better static mate detection and further knowledge about enforceable “mate in X” positions. It is also possible to down-scale and specifically adapt the algorithm for the subgames KBKN, KNKN, KNNK, and the endgame KBKB.

References


MODEL ENDGAME ANALYSIS

G.McC. Haworth, R.B. Andrist
guy_haworth@hotmail.com; rba_schach@gmx.ch

Abstract A reference model of Fallible Endgame Play has been implemented and exercised with the chess engine WILHELM. Various experiments have demonstrated the value of the model and the robustness of decisions based on it. Experimental results have also been compared with the theoretical predictions of a Markov model of the endgame and found to be in close agreement.

Keywords: chess, endgame, experiment, fallibility, Markov, model, theory

1. Introduction

In Haworth (2003), a reference model of fallible endgame play was defined in terms of a spectrum of Reference Endgame Players (REPs) \( R_c \). The REPs are defined as choosing their moves stochastically, using only successor positions’ values and depths from an endgame table (EGT). Exploring here the parameters of the model and various opponent-sensitive uses of the REPs including choice of move, we report on:

a) the robustness of decisions based on the model, given that various parameters of the model may be changed,
b) the apparent competence of reference player \( R_{20} \) in an \( R_{20} - R_\infty \) match,
c) the distribution of game lengths in that match versus Markov theory,
d) the probability of beating a 50-move draw claim versus Markov theory,
e) the apparent competence of carbon and silicon players over the board.

In Section 2, we revisit the basic concepts and theory of the REP model, while in Section 3, we describe the REP implementation in WILHELM (Andrist, 2003). In Sections 4 to 7, we focus on the five topics above. Section 8 summarises and notes some questions arising from this work.
2. The Reference Endgame Player Model

A nominated endgame, e.g., chess’ KQKR, is considered to be a system with a finite set of states \( \{s_i\} \) numbered from 0 to \( ns-1 \).\(^1\) Each state \( s(val, d) \) is an equivalence class of positions of the same theoretical value \( val \) and depth \( d \). Higher-numbered states are assumed to be less attractive to the side to move, which is taken to be White. Thus, for KQKR with the DTC\(^2\) metric, we have \( \text{maxDTCs} \) (1-0) \( n_w = 31 \), (0-1) \( n_B = 3 \), and \( ns = 37 \) states in total:

- \( s_i, i = 0 \): a 1-0 win, i.e., for White, not requiring a winner’s move\(^3\),
- \( s_i, 1 \leq i \leq 31 \): 1-0 wins of depth \( i \),
- \( s_i, i = 32 \): theoretical draw, either in the endgame or a subgame,
- \( s_i, 33 \leq i \leq 35 \): 0-1 wins, i.e., for Black, of depth 36-\( i \)
- \( s_i, i = 36 \): a 0-1 win not requiring a winner’s move.

The REP \( R_c \) in position \( P \) chooses stochastically from moves which each have a probability proportional to a Preference\(^4\), \( S_c(val_s, d_s) \), where \( s \) is the move’s destination state with theoretical value \( val_s \) and win/loss depth \( d_s \). Each move-choice by \( R_c \) is independent of previous move-choices.

We require that \( \{R_c\} \) is a spectrum of players, ranging linearly from the metric-infallible player \( R_\infty \) via the random player \( R_0 \) to the anti-infallible player \( R_-\infty \). To ensure this, the function \( S_c(val, d) \) is required to meet some natural criteria, as described more fully and formally in Haworth (2003) and in Appendix B.

Here, we choose, as an \( S_c(val, d) \) function meeting those criteria:

\[
\begin{align*}
S_c(\text{win}, d) &\equiv (d + \kappa)^c \quad \text{with } \kappa > 0 \text{ to ensure that } S_c \text{ is finite,} \\
S_c(\text{draw}) &\equiv S_c(\text{win}, n_1) \equiv S_c(\text{loss}, n_2) \quad \text{with } n_1 > n_w \text{ and } n_2 > n_B, \\
S_c(\text{loss}, d) &\equiv \lambda \cdot (d + \kappa)^c, \lambda \text{ being defined by } n_1 \text{ and } n_2 \text{ above.}
\end{align*}
\]

This ensures, as required, that \( R_0 \) prefers no move to any other, that \( R_c \) with \( c > 0 \) prefers better moves to worse moves, and that as \( c \to \infty \), the \( R_c \) increase in competence and tend to infallibility in terms of the chosen metric.

Although the \( R_c \) have no game-specific knowledge, the general REP model allows moves to be given a prior, ancillary, weighting \( v_m \) based on such considerations (Jansen, 1992). Thus, \( v_m = 0 \), as used in this paper, prevents a move being chosen and \( v_m > 1 \) makes it more likely to be chosen.

The probability \( T_c(i) \) of moving to state \( s_i \) is therefore:

\[
T_c(i) \equiv S_c(s_i) \cdot \sum \text{moves to state } i \cdot v_m / \sum \text{all moves } v_m \cdot S_c(s_{\text{move}})
\]

---

\(^1\) For convenience, Appendix A summarises the key acronyms, notation, and terms.

\(^2\) DTC = DTC(onversion) = Depth to Conversion, i.e., to mate and/or change of material.

\(^3\) i.e., mate, achieved conversion to won subgame, or loser forced to convert on next move.

\(^4\) For convenience and clarity, the Preference Function \( S_c(val_s, d_s) \) may be signified by the more compact notations \( S_c(val, d) \) or merely \( S_c(s) \) if the context allows.
3. Implementing the REP Model

The second author has implemented in WILHELM (Andrist, 2003) a subset of the REP model which is sufficient to provide the results of this paper.

Ancillary weightings \( \nu_m \) are restricted to 1 and 0. \( \nu_m = 0 \) is, if relevant, applied to all moves to a state \( s \) rather than to specific moves: it can be used to exclude moves losing theoretical value, and/or to emulate a search horizon of \( H \) moves, within which a player will win or not lose if possible.

WILHELM offers five agents based on the REP model: these are, as defined below, the Player, Analyser, Predator, Emulator, and Predictor. A predefined number of games may be played between any two of WILHELM, Player, Predator, Emulator and an infallible player with endgame data. WILHELM also supports the creation of Markov matrices, see Section 5.

3.1 The Player

The Player is an REP \( R_c \) of competence \( c \), and therefore chooses its moves stochastically using a validated (pseudo-)random number generator in conjunction with the function \( S_c(val, d) \) defined earlier.

3.2 The Analyser

Let us imagine that an unknown fallible opponent is actually going to play as an \( R_c \) with probability \( p(d) \cdot dx \) that \( c \in (x, x + \delta x) \): \( \int p(x)dx = 1 \).

The Analyser attempts to identify the actual, underlying \( c \) of the \( R_c \) which it observes. For computational reasons, the Analyser must assume that \( c \) is a value from a finite set \( \{c_j\} \) and that \( c = c_j \) with initial probability \( p_{c_{0,j}} \).

Here, the \( c_j \) are regularly spaced in \([c_{\text{min}}, c_{\text{max}}]\) as follows:

\[
    c_{\text{min}} = c_1, c_j = c_1 + (j-1) \cdot c_\delta \text{ and } c_{\text{max}} = c_1 + (n-1) \cdot c_\delta, \text{ i.e. } c = c_{\text{min}}(c_\delta)c_{\text{max}}.
\]

The notation \( c = c_{\text{min}}(c_\delta)c_{\text{max}} \) is used to denote this set of possible values \( c \). The initial probabilities \( p_{c_{0,j}} \) may be \( 1/n \), the usual ‘know nothing’ uniform distribution, or may be based on previous experience or hypothesis. They are modified, given a move to state \( s_{\text{next}} \), by Bayesian inference:

\[
    T_j(next) = \text{Prob}[\text{move to state } s_{\text{next}} | c = c_j], \quad \text{and}
    \]

\[
    p_{c_{i+1,j}} = p_{c_{i,j}} T_j(next) / \sum_k [p_{c_{i,k}} T_k(next)].
\]

Thus, the new Expected[c] = \( \sum_j p_{c_{i+1,j}} c_j \).

In Subsection 4.1, we investigate what values should be chosen for the parameters \( c_{\text{min}}, c_\delta \) and \( c_{\text{max}} \) so that the errors of discrete approximation are acceptably small.
3.3 The Predator

On the basis of what the Predator has learned from the Analyser about its opponent, it chooses its move to best challenge the opponent, i.e., to optimise the expected value and depth of the position after a sequence of moves. As winning attacker, it seeks to minimise expected depth; as losing defender, it seeks to maximise expected depth. In a draw situation, it seeks to finesse a win.

Different moves by the Predator create different sets of move-choices for the fallible opponent. These in turn lead to different expectations of theoretical value and depth after the opponent's moves.

The predator implementation in Wilhelm chooses its move on the basis of only a 2-ply search. It may be that deeper searches will be worthwhile, particularly in the draw situation.

3.4 The Emulator

The Emulator $E_c$ is conceived as a practice opponent with a ‘designer’ level of competence tailorable to the requirements of the practising player. An REP $R_c$ will exhibit an apparent competence $c'$ varying, perhaps widely, above and below $c$ because it chooses its moves stochastically. In contrast, the Emulator $E_c$ chooses a move which exhibits to an Analyser an apparent competence $c''$ as close to $c$ as possible.

The reference Analyser is defined as initially assuming the Emulator is an $R_x$, $x = 0(1)2c$, where $x = x_j$ with initial probability $1/(2c+1)$.

The Emulator $E_c$ therefore opposes a practising player with a more consistent competence $c$ than would $R_c$, albeit with some loss of variety in its choice of moves. The value $c$ can be chosen to provide a suitable challenge in the practice session.

The practising player may also have their apparent competence assessed by the Analyser.

3.5 The Predictor

The Predictor is advised of the apparent competence $c$ of the opponent. It then predicts how long it will take to win, or what its chances are of turning a draw into a win, using data from an Analyser and from a Markov model of the endgame. This model is defined in Section 5.
4. Robustness of the Model

The two famous Browne-BELLE KQKR exhibition games have already been studied using the REP model (Haworth, 2003). Browne's apparent competence \( c \) was assessed by an \textit{Analyser}, and BELLE's moves as Black were compared with the decisions of a \textit{Predator} using the \textit{Analyser}'s output.

In that analysis, the following six choices were made:

- \( c_{\text{min}} = 0, c_\delta = 1, c_{\text{max}} = 50; \) all \( c_j \) were deemed equally likely,
- \( \kappa = 0^+ \) (i.e., arbitrarily small, effectively zero) and metric = DTC.

The following question therefore arises: to what extent are the conclusions of the \textit{Analyser} and the choices made by the \textit{Predator} affected by these six choices? Our first studies addressed this question.

4.1 The Effect of Numerical Approximation

Browne-BELLE game 1 was first reanalysed, this time with \( \kappa = 1 \), and:

\[ c_{\text{min}} = 0, c_{\text{max}} = 50 \text{ and } c_\delta \text{ in turn set to } 0.01, 0.1, 1, 2, 5 \text{ and } 10. \]

Figure 1 takes the \textit{Analyser} with \( c_\delta = 0.01 \) as a benchmark, and shows how the choice of \( c_\delta \) affected the \textit{Analyser}'s inferences during play.

It may be shown the \textit{Analyser}'s Bayesian calculation is a discrete approximation to a calculable integral: the theory of integration therefore guarantees that this calculation will converge as \( c_\delta \to 0 \). We judge that the error is ignorable with \( c_\delta = 1 \) and that no smaller \( c_\delta \) is needed.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Differences in \( c \)-estimation, relative to the \( c_\delta=0.01 \) estimate.}
\end{figure}
The analysis of the game was then repeated with:

c_\delta = 1, c_{\min} = 0 and c_{\max} in turn set to 100, 90, 80, 70, 60, 50, 40 and 30.

Again, intuitively, we would expect the error introduced by a finite $c_{\max}$ to reduce as $c_{\max} \to \infty$. Figure 2 shows that this is indeed the case and that, with Browne's apparent $c = 20$, $c_{\max} = 50$ is conservative enough. However, it may need to be larger for easier endgames.

We assume that our opponent has positive apparent competence $c$ and that the Analyser is correct in taking $c_{\min} = 0$ as a lower bound on $c$.

### 4.2 The Effect of $\kappa$

Given the requirements on $S_c(val, d)$, it may be shown$^5$ that, as $\kappa$ increases, $R_c$ progressively loses its ability to differentiate between better and worse moves, that $R_c$'s expectation of state and theoretical value do not improve and that $R_c \to R_0$. Thus, for a given set of observations, if the Analyser assumes a greater $\kappa$, it will infer an increasing apparent competence $c$.

In this paper, we choose a fixed $\kappa = 1$ throughout, as it were, recognising the next move in the line contemplated. We have not tested the effect of different $\kappa$ on a Predator's choices of move, but assume it is not great. There seems little reason to choose one value of $\kappa$ over another.

$^5$ The proof is by elementary algebra and in the style of Theorem 3 (Haworth, 2003).
4.3 The Effect of the Initial Probability Assumption

The usual, neutral, initial stance is a *know nothing* one, assuming that $c$ is uniformly distributed in a conservatively-wide interval $[c_{\text{min}}, c_{\text{max}}]$. However, it is clear that had BELLE been using the REP model, it could have started game two with its perception of Browne as learned from game one, just as Browne started that game with his revised perception of KQKR. Also, one might have a perception of the competence $c$ likely to be demonstrated by the opponent with the given endgame force – and choose this to be the mid-point of a $[c_{\text{min}}, c_{\text{max}}]$ range with a normal distribution.

Bayesian theory, see Subsection 3.2, shows that the initial, assumed non-zero probabilities continue to appear explicitly in the calculation of subsequent, inferred probabilities. We therefore conclude that initial probabilities have some effect on the inferred probabilities.

4.4 The Effect of the Chosen Metric

The metric Depth to Conversion (DTC) was chosen because *conversion* is an obvious intermediate goal in most positions. The adoption of DTC is however a chessic decision.

Our analysis of the Browne-BELLE games shows that the *Predator* would never have made a DTC-suboptimal move-choice for Black. It is reasonable to assume that, had DTM(ate) been the chosen metric, it would never have chosen a DTM-suboptimal move.

However, different metrics occasionally define different subsets of moves as metric-optimal. Where this occurs, the *Predator* might well choose a different move in its tracking of the Browne-BELLE games.

5. A Markov Model of the Endgame

Let us assume that the Preference Function $S_c(val, d)$ is fixed, e.g., as the function defined here with $\kappa = 1$.

Given a position $P$ in state $s_i$, we can calculate the probability of $R_c$ choosing move $m$ to some position $P'$ in state $s_j$. We may therefore calculate the probability, $T_c(j)$ of moving from position $P$ to state $s_j$. Averaging this across the endgame over all such positions $P$ in state $s_i$, we may derive the probability $m_{ij}$ of a state transition $s_i \rightarrow s_j$ assuming initial state $s_i$.

The $\{m_{ij}\}$ define a Markov matrix $M_c = [m_{ij}]$ for player $R_c$. This matrix, and the predictions which may be derived from it, provide a characterisation of the endgame as a whole.

Let us assume that the initial position is 1-0, in state $s_i$, and that $R_c$ does not concede the win. From the matrix, we may derive predictions such as:
the probability of \( R_e \) winning on or before move \( m \),

- the expected number of moves required for \( R_e \) to achieve the win.

This is \( L_i \) in the solution of \((I - M_e)L = U^k\), q.v. (Haworth, 2003).

These theoretical predictions have been computed and are compared with the results of the extensive experiment described in the next section.

6. An Experiment with \( R_{20} \)

Echoing Browne-BELLE, a model KQKR match was staged between the fallible attacker \( R_{20} \) and the infallible defender \( R_\infty \). It was assumed that \( R_{20} \) would not concede the win but eventually secure it as theory predicts. The game-specific repetition and 50-move drawing rules were assumed not to be in force. Table 1 summarises the results of this experiment.

1,000 games were played from each of the two maxDTC KQKR positions used in the Browne-BELLE match. Games ended when conversion was achieved by White. The purpose of the experiment was to observe:

- the distribution of the \( c \) inferred by an Analyser\(^7\) at the end of each game with the assumed probability of \( c_i \) set to 1/51 at start of each game,

- the distribution of the lengths of the games, and

- the trend in the Analyser's inferred \( c \), ignoring game-starts after the first.

<table>
<thead>
<tr>
<th>KQKR: ( R_{20} - R_\infty )</th>
<th>Position 1</th>
<th>Position 2</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min., end-of-game apparent ( c )</td>
<td>15.06</td>
<td>14.73</td>
<td>14.73</td>
</tr>
<tr>
<td>Max., end-of-game apparent ( c )</td>
<td>35.66</td>
<td>40.71</td>
<td>40.71</td>
</tr>
<tr>
<td>Mean, end-of-game apparent ( c )</td>
<td>21.318</td>
<td>21.620</td>
<td>21.469</td>
</tr>
<tr>
<td>St. Dev., end-of-game apparent ( c )</td>
<td>3.345</td>
<td>3.695</td>
<td>3.524</td>
</tr>
<tr>
<td>St. Dev of the Mean apparent ( c )</td>
<td>0.106</td>
<td>0.117</td>
<td>0.079</td>
</tr>
<tr>
<td></td>
<td>Mean ( c ) - 20/Stddev_mean</td>
<td>12.43</td>
<td>13.85</td>
</tr>
<tr>
<td></td>
<td>Min. moves, ( m ), to conversion</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>Maximum moves, ( m )</td>
<td>395</td>
<td>325</td>
</tr>
<tr>
<td></td>
<td>Mean moves, ( m )</td>
<td>96.88</td>
<td>94.31</td>
</tr>
<tr>
<td></td>
<td>St. Dev., ( m )</td>
<td>102.951</td>
<td>102.273</td>
</tr>
<tr>
<td></td>
<td>St. Dev., mean of ( m )</td>
<td>3.256</td>
<td>3.234</td>
</tr>
</tbody>
</table>

\( \text{Table 1. Statistical Analysis of the 2,000-game experiment.} \)

\(^6\) \( I \) is the Identity matrix; \( U \) is a vector where each element is the unit '1'.

\(^7\) using \( c_{\min} = 0 \), \( c_\delta = 1 \) and \( c_{\max} = 50 \) as found adequate in Section 4.1.
6.1 $R_{20}$'s Apparent $c$ after One Game

Figure 3 shows the distribution of the apparent $c$ as inferred at the end of each, single game: the mean $c$ is $21.50 \pm 0.08^8$. This rather surprised us, being more distant from the actual $c = 20$ than expected. The reason is that the mean of \{end-of-game estimated $c$\} is not statistically the best way to estimate the underlying $c$, a task we revisit in Subsection 6.4.

![Figure 3. Distribution of apparent $c$ as inferred after one game.](image)

6.2 Game-Length Statistics

Starting from the two positions with (maximum) DTC depth 31, and taken over the 2,000 games, the mean number of moves required for conversion is $95.60 \pm 2.29$. Figure 4 shows the distribution of the experiment’s game lengths in comparison with the predictions of the Markov model.

Figure 5 shows the Markov-model predictions for the expected number of moves to conversion, for $c = 20, 21, 22$ and starting at any depth. Note that it shows that the main barriers to progress seem to be between depths 17 and 26 rather than at the greatest depths.

From depth 31, the moves predicted are $97.20$ for $c = 20$, $83.70$ for $c = 21$ and $74.16$ for $c = 22$. The experimental results are therefore in close agreement with these predictions, indicating a $c$ of $\sim 20.1$.

---

8 Mean end-of-game apparent $c$ is still 21.04 when the Analyser’ $c_{\text{max}}$ is 30 rather than 50.
The games were played without the 50-move rule but the Markov model allows us to calculate the probability of winning from depth $d$ on or before move 50, before a possible draw claim by the opponent. It is the probability of being in state 0 after 50 moves, namely the element $M_c^{50}[d, 0]$ of $M_c^{50}$. 

6.3 The Probability of Winning

---

**Figure 4.** Distribution of game lengths in the $R_c-R_{\infty}$ KQKR match.

**Figure 5.** Expected [moves to conversion in an $R_c-R_{\infty}$ KQKR game].
Figure 6 gives these probabilities for \( c = 20, 21, \) and 22 and for all initial depths. For \( c = 20 \) and initial depth 31, this is 12.67%, a figure reached after 55 moves in the 2,000 game experiment.

![Graph showing the probability of \( R_c \) winning an \( R_c-R_{\alpha} \) KQKR game in 50 moves.

Figure 7. Analyser error in \( c \)-estimate versus number of games analysed.
6.4 Analyzing $R_e$'s competence $c$

The mean of the 2,000 end-of-game apparent $c$ values is not actually the best estimate of $R_e$'s underlying $c$.

The reason is that the 2,000 games may be seen as some 191,200 independent move-choices by the $R_e$. There is no need to associate the *know nothing* uniform distribution probabilities with the possible $c_j$ more than once. In fact, to do so is to interrupt the Bayesian inference processor of the Analyser and to negate what the Analyser has learned from previous games about the non-uniform distribution of probabilities of the candidate $c_j$.

Figure 7 shows the Analyser's perception of $c$ approaching the correct value of 20 as it works through the 2,000 games. Even starting with an estimate of $c = 25$, it is accurate to 0.1 after examining 6,000 moves.

7. Apparent Competence of Players

The apparent competence of both carbon and silicon players has been calculated for some published games. The initial assumptions differed slightly from those of Haworth (2003): here, WILHELM was set to analyse with candidate $c = 0$ (0.01) 50 and $\kappa = 1$ and the results are listed in Table 2, showing depth conceded by both sides, net progress and apparent $c$.

Some background to the games may help put the figures in context. The two Browne-BELLE games (Fenner, 1979; Haworth, 2003; Jansen, 1992a; Levy and Newbom, 1991) are the famous demonstration that the 'easy' KQKR endgame is not so easy to win. Gelfand-Svidler was a tie-breaker rapid-play game played under extreme time pressure. Pinter-Bronstein has been extensively analysed by Roycroft (1988). Timman consulted extensively in a prior adjournment (Breuker et al., 1992). FRITZ (Heise, 2002) played itself with only 3-to-4-man EGTs in an Intel-AMD duel. Lengyel lost the draw three times before our analysis begins (Levy, 1972a,b, 1992).

<table>
<thead>
<tr>
<th>#</th>
<th>Profile</th>
<th>White</th>
<th>Black</th>
<th>Res.</th>
<th>Year</th>
<th>#m</th>
<th>depth lost</th>
<th>depth gain</th>
<th>Final $c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>KQKR</td>
<td>Browne</td>
<td>BELLE 1</td>
<td>=</td>
<td>1978</td>
<td>45</td>
<td>27</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>KQKR</td>
<td>Browne</td>
<td>BELLE 2</td>
<td>1-0</td>
<td>1978</td>
<td>50</td>
<td>19</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>KRKQ</td>
<td>Gelfand</td>
<td>Svidler</td>
<td>=</td>
<td>2001</td>
<td>50</td>
<td>37</td>
<td>79</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>KNKBB</td>
<td>Pinter</td>
<td>Bronstein</td>
<td>=</td>
<td>1977</td>
<td>50</td>
<td>60</td>
<td>95</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>KBBKN</td>
<td>Popovich</td>
<td>Korchnoi</td>
<td>=</td>
<td>1984</td>
<td>31</td>
<td>74</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>KBBKN</td>
<td>Timman</td>
<td>Speelman</td>
<td>1-0</td>
<td>1992</td>
<td>25</td>
<td>36</td>
<td>44</td>
<td>33</td>
</tr>
<tr>
<td>7</td>
<td>KNKBB</td>
<td>FRITZ</td>
<td>FRITZ</td>
<td>=</td>
<td>2002</td>
<td>49</td>
<td>169</td>
<td>210</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>KQKQN</td>
<td>Lengyel</td>
<td>Levy</td>
<td>0-1</td>
<td>1972</td>
<td>14</td>
<td>7</td>
<td>16</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 2. Apparent Competence of Players.
Even after noting that the values $c$ are not necessarily the player’s true equivalent $c$, and are meaningful only in relative rather than absolute terms, the performances of Browne and Timman stand out. FRITZ trades major depth with its clone-opponent and clearly misses the withheld perfect information. The time constraints of Rapid Play, and even third-phase 30'/game play in classical chess, mitigate against quality endgame play – arguably a loss to the world of chess.

8. **Summary**

We have examined the utility of a reference model of Fallible Endgame Play by both experiment and theory, using both a comprehensive REP implementation in WILHELM and Markov methods. Various demonstrations have shown opportunities for exploiting the model, and the robustness of decisions based on it. Experimental results have also been compared with the Markov predictions, with which they agree closely.

Experiments which remain to be carried out include:
- infallible White attacking fallible Black in a drawn position  
  e.g., in KBBKN, KNNKP, KNPKN, KQNKQ, KQPKP, or KRBRK,
- infallible Black pressing for a draw in a lost position  
  this requires additional EGT data on draws forced in $d$ moves,
- a more insightful Predator searching more than $2p$ plies ahead, and
- use of the Emulator as a training partner for human players.

The REP model may be extended to other games where EGTs may be computed – to convergent games such as Chinese Chess, 8×8 checkers, International Draughts, and in principle if not in practice, to divergent placement games such as Hex and Othello.

If a search method can propose what it considers the best few moves in a position, each evaluated on an identical basis and therefore comparable, the concept of a stochastic player may be applied more generally than to just endgames for which perfect information is available.

**Acknowledgements**

We thank the ACG10 conference sponsors, organizers, and referees for their considerable support and advice on this paper. We thank Walter Browne for his excellent sporting example in facing up to Ken Thompson’s silicon beast in the KQKR match of 1979. Also, we thank those who have generated and/or made available definitive EGT data over the years.
References

Tamplin, J. (2001). Private communication of some pawnless Nalimov-compatible DTC EGTS.

Appendix A: Acronyms, Notation and Terms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyser</td>
<td>an agent identifying a fallible opponent as an $R_c$ player</td>
</tr>
<tr>
<td>$c$</td>
<td>the competence index of an REP</td>
</tr>
<tr>
<td>$c\delta$</td>
<td>the difference between adjacent $c_i$ assumed by the Analyser</td>
</tr>
<tr>
<td>$c_{\max}$</td>
<td>the maximum $c$ assumed possible by the Analyser</td>
</tr>
<tr>
<td>$c_{\min}$</td>
<td>the minimum $c$ assumed possible by the Analyser</td>
</tr>
<tr>
<td>$d$</td>
<td>the depth (of win or loss) of a position in the chosen metric, e.g. DTC</td>
</tr>
<tr>
<td>DTC</td>
<td>Depth to Conversion, i.e. to change of material and/or mate</td>
</tr>
<tr>
<td>DTM</td>
<td>Depth to Mate</td>
</tr>
<tr>
<td>Emulator</td>
<td>an agent, $E_c$, choosing moves to best exhibit apparent competence $c$</td>
</tr>
<tr>
<td>Horizon</td>
<td>a search limit, within which $R_c$ will win or not lose if possible</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>$\kappa &gt; 0$ ensures that $(d + \kappa)^c$ is finite</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>a scaling factor, matching the probability of loss to that of a draw</td>
</tr>
<tr>
<td>$L_i$</td>
<td>expected length of win (to conversion in winner’s moves) from depth $i$</td>
</tr>
<tr>
<td>maxDTC</td>
<td>maximum DTC (depths)</td>
</tr>
<tr>
<td>$M_c$</td>
<td>a Markov matrix $[m_{ij}]$</td>
</tr>
<tr>
<td>$m_{ij}$</td>
<td>the probability, averaged over the endgame, that $R_c$ in state $s_i$ moves to $s_j$</td>
</tr>
<tr>
<td>metric</td>
<td>a measure of the depth of a position, usually in winner’s moves</td>
</tr>
<tr>
<td>$n$</td>
<td>the number of different $c_i$ assumed by an Analyser</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$n_1 &gt; n_w$, ensures that draws are less preferable than wins</td>
</tr>
<tr>
<td>$n_2$</td>
<td>$n_2 &gt; n_B$, ensures that draws are more preferable than losses</td>
</tr>
<tr>
<td>$n_B$</td>
<td>the number of ‘Black win’ states</td>
</tr>
<tr>
<td>$n_w$</td>
<td>the number of ‘White win’ states</td>
</tr>
<tr>
<td>$n_s$</td>
<td>the number of states for a chosen endgame and depth metric</td>
</tr>
<tr>
<td>$p(x)\cdot\delta x$</td>
<td>the probability that $R_c$’s $c \in [x, x + \delta x]$</td>
</tr>
</tbody>
</table>
Model Endgame Analysis

\( p_{ij} \) the \textit{a priori} (before a move) probability that the unknown \( c \) is \( c_j \)

\( p_{ij} \) the probability, inferred after the \( i \)th move, that the unknown \( c \) is \( c_j \)

\textbf{Player} an \( R_c \), choosing its moves stochastically with Preference Function \( S_c \)

\textbf{Predator} an agent, choosing the best move possible on the basis of an opponent-model

\textbf{Predictor} an agent predicting the longer term prospects of a result from Markov theory

\textbf{REP} Reference Endgame Player

\( R_0 \) the REP which prefers no move to any other

\( R_c \) an REP of competence \( c \)

\( R_{\infty} \) the player which plays metric-optimal moves infallibly

\( s \) endgame state

\( s_i \) (endgame) state \( i \)

\( S_c(val, d) \) the Preference Function for REP \( R_c \), a function of destination value and depth

\( S_c(val, d) \) a convenient contraction of \( S_c(val, d) \)

\( S_c(s) \) a more convenient contraction of \( S_c(val, d) \)

\( T_c(i) \) the probability that \( R_c \) moves to state \( i, s_i \)

\( \text{val} \) the theoretical value of a position, i.e., \textit{win}, \textit{draw} or \textit{loss}

\( v_m \) a weighting that may be given to a move on chessic grounds

\textbf{Appendix B: Preference Functions}

We require that the set \( \{ R_c \} \) is in fact a linear, ordered spectrum of \( R_c \) players such that:

- for \( R_0 \), all moves are equally likely,
- \( 'R_{\infty}' = \lim_{c \to \infty} R_c \) exists and is the infallible player choosing metric-optimal moves,
- \( 'R_{\infty}' = \lim_{c \to \infty} R_c \) exists and is the anti-infallible player choosing anti-optimal moves,
- \( c_2 > c_1 \Rightarrow R_{c_2} \)'s expectations of successor state, i.e. \( E[s] \), are no worse than \( R_{c_1} \)'s,
- \( c_2 > c_1 \Rightarrow R_{c_2} \)'s expectations of theoretical value, i.e. \( E[\text{val},] \), are no worse than \( R_{c_1} \)'s.

The following requirements on \( S_c(val, d) = S_c(s) \) are natural ones and sufficient to ensure the above, as proved in Haworth (2003):

- \( S_c(s) \) is finite and positive: no move has zero or infinite preference for finite \( c \),\(^9\)
- \( S_0(s) \) is a constant,
- for some \( n_1 > n_2 \) and \( n_2 > n_B \), \( S_c(\text{draw}) = S_c(\text{win}, n_1) = S_c(\text{loss}, n_2) \),
- \( F_j(c) = S_c(s_{j+1})/ S_c(s_j) \) decreases as \( c \) increases: \( \lim_{c \to \infty} F_j(c) = 0 \) and \( \lim_{c \to \infty} 1/F_j(c) = 0 \),
- for \( c \neq 0 \), sign(\( c \))-\( S_c(s_j) \) decreases (↓) as \( j \) increases (↑),
- for \( c > (<) 0 \), \( W_c(d) = S_c(\text{win}, d)/ S_c(\text{win}, d+1) \) ↓ (↑) as \( d \) ↑ and \( \lim_{d \to \infty} W_c(d) = 1 \),
- for \( c > (<) 0 \), \( L_c(d) = S_c(\text{loss}, d+1)/ S_c(\text{loss}, d) \) ↓ (↑) as \( d \) ↑ and \( \lim_{d \to \infty} L_c(d) = 1 \).

The net effect is that:

- the spectrum of \( R_c \) is centred as required on the random player, \( R_0 \),
- the \( R_c \) with \( c > 0 \) prefer better moves to worse moves,
- the \( R_c \) demonstrate increasing apparent skill as \( c \to \infty \),
- \( R_c \) can be arbitrarily close to being the metric-infallible player for finite \( c \)
- as \( d \to \infty \), \( R_c \) discriminates less between a win (or loss) of depth \( d \) and one of depth \( d+1 \).

\(^9\) Hence the requirement that \( \kappa > 0 \), to accommodate the case of \( d = 0 \) in \((d + \kappa)^c\).
CHESS EN DGAMES: DATA AND STRATEGY

J.A. Tamplin, G.M'C. Haworth
jat@jaet.org; guy_haworth@hotmail.com, http://http://www.jaet.org/jat/

Abstract While Nalimov's endgame tables for Western Chess are the most used today, their Depth-to-Mate metric is not the only one and not the most effective in use. The authors have developed and used new programs to create tables to alternative metrics and recommend better strategies for endgame play.

Keywords: chess, conversion, data, depth, endgame, goal, move count, statistics, strategy

1. Introduction

Chess endgames tables (EGTs) to the 'DTM' Depth to Mate metric are the most commonly used, thanks to codes and production work by Nalimov (Nalimov, Haworth, and Heinz, 2000a,b; Hyatt, 2000). DTM data is of interest in itself, even if conversion, i.e., change of force, is usually adopted as an interim objective in human play. However, more effective endgame strategies using different metrics can be adopted, particularly by computers (Haworth, 2000, 2001). A further practical disadvantage of the DTM EGTs is that, with more men, DTM increases and file-compression becomes less effective.

Here, we focus on metrics DTC, DTZ\(^1\) and DTZ\(_{50}\)^{2}; the first two were previously used by Thompson (1986, 2000) and Wirth (1999). New programs by Tamplin (2001) and Bourzutschky (2003) have enabled a complete suite of 3-to-5-man DTC/Z/Z\(_{50}\) EGTs to be produced.

Section 2 outlines these two new algorithms. Sections 3 to 5 review the new DTC, DTZ and DTZ\(_{50}\) data tabled in the Appendix. Finally, improved endgame strategies are recommended for the 50-move context:

\(^1\) DTC = Depth to Conversion, i.e., to force change and/or mate.

\(^2\) DTZ\(_k\) = Depth to (Move-Count) Zeroing (Move), i.e., to P-push, force change and/or mate.

DTZ\(_k\) = DTZ, but draw if the 'win' can be pre-empted by a \(k\)-move draw claim.
2. New Approaches to EGT Generation

Below we briefly describe two new approaches to EGT generation. The first one is described adequately in the literature; the second so far not.

2.1 Tamplin’s Wu-Beal Code

Tamplin (2001) combined the Wu-Beal (2001a,b) algorithm with Nalimov indexing in a new code whose objectives were primarily Nalimov-compatibility, simplicity, maintainability and portability. Most pawnless 3-to-5-man DTC EGTs were generated, the new code including an inverse-index function mirroring Nalimov’s index function.

2.2 Bourzutschky’s Modified-Nalimov Code

Bourzutschky (2003) modified Nalimov’s DTM-code to enable it also to generate EGTs to metrics \( \text{DTC}_k \) and \( \text{DTZ}_k \). This involved generalising some DTM-specific aspects of the algorithm, as well as the obvious changes to the iterative formula for deriving depth. For DTC, the code retains the efficiencies of the DTM-code while requiring maxDTC rather than maxDTM cycles. Because EGT generation to the DTZ metric has not yet been implemented generically as a sequence of sub-EGT generations, each based on a fixed pawn structure, this is not the case for \( \text{DTZ}_k \) computations. These can also require somewhat more than DTC cycles but the difference is insignificant.

3. The DTC Data

DTC EGTs are interesting, not only for completeness, but because conversion is an intuitively obvious objective and the DTC EGTs document precisely the phase of play when the material nominated is on the board.

The remaining 3-to-5-man DTC EGTs were generated. Table 1 in the Appendix lists for each endgame the number of positions of maxDTC, wtm/btm and 1-0/0-1. The ICGA (2003) website provides further data, including %-wins, illustrative maxDTC positions and DTC-minimaxing lines. Because there are many wins in 1, the % of positions won does not characterise well the presence of wins in an endgame. Similarly, maxDTC is not a good indicator. We therefore suggest a new characteristic,

\[
\text{Win-Presence} = \% \text{ of positions won} \times \text{(Average DTC of Win)}
\]

This is not unduly affected by the usual peak of wins in 1 or by the long tail of deep wins, and is in fact related to the number of moves for which a win is present on the board.
3.1 A Review of the DTC Data

A first housekeeping point to be made is that this data often differs from Wirth's data (Wirth and Nievergelt, 1999; Tamplin, 2003). The explanation is simple. First, Wirth has exactly one representative of each equivalence class of positions, including the harder case of both Kings being on a1-h8. Nalimov would count \{wKc3Qb3(c2)/bKa1\} as two positions rather than one.

Second, Wirth's code, based on the inherited RETROENGINE, assumes that all conversions are effected by the winner. This is not so: the loser is sometimes forced to convert to loss, e.g., \{wKe1Qb1Rf1/bKa1\}, in which case Wirth's depth is too great by one.

Tamplin's (2003) and Bourzutschky's (2003) codes both measure depth consistently in winner's moves. Also, they do not allow 'realistic' but voluntary conversions, e.g., \{wKe1Qf1Rb1/bKa1\}, by the loser, a feature of Thompson's original DTC EGT code (Thompson, 1986) which chose to move to the position with greatest DTC even if a capture was involved.

The sub-6-man compressed DTC EGTs are 62.1% the size of the DTM EGTs, usefully saving 2.8GB disc space.

The maxDTC=114 wins in KNNKP and KQPKQ are already known. KBNK wtm scores the highest in Win-Presence terms: maxDTC = 33, average DTC = 24.68 and 99.51% of positions are 1-0 wins.

4. The DTZ Data

The DTZ metric is necessary if the length of the current phase of play is to be guarded in the context of chess' k-move rule, k currently being 50. It was used pragmatically by Thompson (1986) to compute the KQPKQ and KRPKR EGTs when RAM was relatively scarce.

Bourzutschky (2003) generated some DTZ EGTs where maxDTZ > 50 and Tamplin (2003) completed the sub-6-man DTZ EGT suite. The computation continues to be a major feat as it cannot currently use Nalimov's bitvector-based algorithm which reduces RAM requirements by a factor of 4 to 16.

Table 2 in the Appendix lists the results which differ from the DTC data. KNNKP with maxDTZ = 82 features the deepest endings. DTZ EGTs are commendably compact relative to DTM and DTC EGTs. The KPPPK wtm DTZ EGT is an extreme example, being only 2% the size of the DTM EGT. In total, the sub-6-man compressed DTZ EGTs are 52.9% the size of the DTM EGTs, usefully saving some 3.5GB of disc space.
5. The DTZ$_{50}$ Data

Bourzutschky (2003) and Tamplin (2003) also generated DTZ$_{50}$ EGTs, not only for those cases where maxDTZ > 50, but for endgames directly or indirectly dependent on these as illustrated in Figure 1. The DTZ$_{50}$ metric rates as wins only those positions winnable against best play given the 50-move rule. In Figure 1, endgames for which EZ and EZ$_{50}$ are potentially but not actually different are in brackets, and dotted lines indicate that no 50-move impact emanates from or feeds back to them.

The sub-6-man compressed DTZ$_{50}$ EGTs are 49.8% the size of the DTM EGTs. Table 3 in the Appendix lists 3-to-5-man DTZ$_{50}$ EGT data for endgames where DTZ$_{50}$ $\neq$ DTZ and Table 7 gives examples of positions affected. Table 6 summarises 50-move impact, minimal for KNPKQ, considerable for KBBKNN and KNNKP.

![Diagram](image)

*Figure 1.* Endgames with EZ$_{50}$ $\neq$ EZ.

If KwKb is an endgame with wtm and btm 1-0 wins impacted by the 50-move rule, KwxB and KwKby are also impacted by the rule. This observation, coupled with Thompson’s DTC results (Tamplin and Haworth, 2001) and the DTM results of Nalimov (Hyatt, 2000) and Bourzutschky (2003) indicate that many 6-man endgames are affected. Tamplin (2003) has computed some of these 6-man endgames’ EGTs to the DTZ and DTZ$_{50}$ metrics.

In contrast with KNNKP, KBBKNN has the majority of its wins frustrated, and few wins can be retained by deeper strategy in the current phase. There are significant percentages of frustrated 0-1 wins in KBBBKNQ, and of delayed 1-0 wins in KBBBKN and KBBNKN.

Elsewhere, there is only the merest hint of the 50-move impact that might follow and we would expect that hint to become fainter as the number of men increases.
6. **Endgame Strategies**

Let $dtx$ be the depth by, and $Ex$ an EGT to, the metric $DTx$. Let $Sx^-$ be an endgame strategy minimising $dtx$, e.g., $SZ^-$, or $SZ_{50}^-$, and let $Sx^+$ be a strategy maximising $dtx$. Further, let $SZ^0$ be an endgame strategy *guarding* the length of the current phase in the context of a $k$-move rule and a remaining $mleft$ moves before a possible draw claim. By definition, if $dtx > mleft$, $Sx^0 \equiv Sx^-$. 

Let $S_{1,2,3}$ be an endgame strategy using strategies $S_1$, $S_2$, and $S_3$ in turn to subset the choice of moves, e.g., $SZ_{50}^0$ $M^-'Z$ which safeguards current phase length and 50-move wins, and then minimises $dtm$ and $dtz$ in turn.

As conjectured by Haworth (2000), KQPKQ and KBBKNN provide positions where all combinations of $SC^-$, $SM^-$ and $SZ^-'$ fail to safeguard a win available under the 50-move rule: the examples here were found by Bourzutschky (2003). Similar positions for other endgames were found by Tamplin (2003). Some strategy-driven lines are listed in Appendix 1 after Table 5.

6.1 **New Endgame Strategies**

$SZ_{50}^-$ wins any game winnable against best play under the 50-move drawing rule. Here, we suggest ways to finesse wins against fallible opposition. If the current phase of play is not unavoidably overlong, strategy $SZ_{50}^0$ $Z^-$, effectively $SZ^0 \equiv SZ^-$, completes it without a draw claim.

For positions where $DTZ_{50}$ indicates draw, the table $EZ_{50}$ can be supplemented by the position’s $DTR^3$ value. Let this hybrid table be $EH_{50}$, implicitly defining metric $DTH_{50}$. Note that $EZ_{50}$ is visible within $EH_{50}$. Since the intention is to use $EH_{50}$ only in conjunction with $EZ$, let the table $E(H/Z)_{50}$ $Z \equiv \{\delta(DT(H/Z)_{50}, DTZ)\}$, giving a compact encoding$^4$ of $E(H/Z)_{50}$ decodable with the use of $EZ$. With $E\delta Z_{50} = \Phi$ if $EZ_{50} \equiv EZ$, sub-6-man compressed $E\delta Z_{50}Z$ EGTs are only 0.7% the size of the DTM EGTs.

The strategy $SZ^0 H_{50}^-$ guards the length of the current phase, wins all games which are wins under the 50-move rule, and minimaxes $DTR$, but only tactically, when the 50-move rule intervenes.

In position NN-P3, $SZ^0 H_{50}^-$ makes the optimal move-choice$^5$. In contrast, $SZ_{50}^-'$ can, and $Sc \ (\sigma \equiv C^-, M^-, Z, Z'Z_{50}^-'Z)$ does, concede $DTR$ depth. However, $SZ^0 H_{50}^-$ has two flaws, the first being a major one. It can draw by repeating positions, e.g., position NN-P4$^6$. $SZ^0 H_{50}^-$ should therefore be augmented by as deep and perceptive a forward search as possible, denoted here by ‘*’ as in $SZ^0 H_{50}^-'$.

---

3 $DTR = Depth by The Rule$ (Haworth, 2000, 2001), i.e. the minimum $k$ s.t. $DTZ_k$ is a win.

4 We chose $0 \equiv \text{“EZ code = Ex code”}$, $1 \equiv \text{“new EZ$_{50}$ draw”}$, $\delta+1 \equiv \text{“0 < DTX – DTZ = $\delta$”}$.

5 $SZ^0 H_{50}^- - SH_{50}^-$: 1. Nb1+ Kc4'. White retains $DTR=51$ and converts in 31 moves.

6 NN-P4, $SZ^0 H_{50}^- - SH_{50}^-$: 1. Nd5+? Kc4' 2. Ndc3 Kb4' {NN-P4 repeated}. 


If position NN-P4, with \( dtz_{51} = 25 \), has just 25 moves left in the phase, it also shows \( SZ_{H_{50}} \) failing to achieve minimal DTR. The move \( Nd5+ \) is optimal for \( SH_{50} \) but \( DTZ_{51} \)-suboptimal, a fact not visible in the EGT \( EH_{50} \). After \( Nd5+ \), \( SZ^0 \) limits the move choice and puts a DTR of 51 out of reach. Again, forward search helps, this time aiming to control DTR.

Any strategy can be sharpened by the opponent sensitivity of an adaptive, opponent model (Haworth, 2003; Haworth and Andrist, 2003).

7. EGT Integrity

All EGT files were given md5sum signatures to guard against subsequent corruption. The EGTs were checked for errors in various ways.

- \( DTx \) EGTs \( \{Ex\} \), \( x = C, Z \) and \( Z_{50} \), verified by Nalimov’s standard test.
- consistency of the \( \{E(C/M/Z)\} \) EGTs confirmed theoretical values found identical with \( dtm \geq dtc \geq dtz \).
- DTC EGT statistics were also found compatible with those of Wirth.
- consistency of the \( \{EZ_{50}\} \) and \( \{EZ\} \) EGTs confirmed linear checks confirm \( EZ_{50} = EZ \) except for known subset, values identical with \( dtz_{50} \geq dtz \), or ‘EZ’ win/loss an ‘EZ_{50}’ draw.

8. Summary

This paper records the separate initiatives of Tamplin (2003) and Bourzutschky (2003) in creating new codes capable of generating non-DTM EGTs. It also reviews the new DTC/Z/Z_{50} data produced by the combination of these codes. The DTC, DTZ and DTZ_{50} EGTs (EC, EZ and EZ_{50}) are increasingly compact compared to the DTM EGTs, an incidental but practical benefit with 3-to-6-man DTM EGTs estimated to be 1 to 2 TB in size.

Together, the sub-6-man compressed EZ and E\(\delta\)Z_{50}Z EGTs are 53.6% the size of the EM EGTs. To date, the equivalent 6-man EGTs are 63.8% the size of their EM EGT counterparts but these do not yet involve Pawns.

Although the computation of DTR data remains a future challenge, table \( EZ_{50} \) may in principle be augmented by DTR values where \( dtr > 50 \) to give table \( EH_{50} \). This table may be used to minimise \( dtz_{50} \) when \( dtz_{50} \leq 50 \), and to minimax \( dtr \) with the assistance of forward-search when \( dtr \geq 50 \).

Clearly, there are more effective and efficient endgame strategies than the commonly used SM\(^*\). It is recommended that \( SZ^oM^*, SZ^oZ_{50}^*Z^*(^*) \), \( SZ^oZ_{50} \), \( ZH_{50}^*Z^* \), \( SZ^oH_{50}^* \) and perhaps other strategies are considered, and that the EZ, E\(\delta\)Z_{50}Z and E\(\delta\)H_{50}Z EGTs are made available to enable their use.
Acknowledgements

We thank Eugene Nalimov for the public 2001 version of his code, and Marc Bourzutschky for modifying it to multi-metric form. Without his achievement, the work reported here would not have been possible. Marc also championed the merits of DTZ_{50}. We thank Rafael Andrist (2003) for a ‘multi-metric’ WILHELM which greatly helped validate and data-mine the EGTs. Finally, we thank those associated with ACG10 for their support.

References

Appendix: Chess Endgame Data and Examples

<table>
<thead>
<tr>
<th>Name</th>
<th>GBR</th>
<th># w-b</th>
<th>wtm</th>
<th>btm</th>
<th>wtm</th>
<th>btm</th>
<th>wtm</th>
<th>btm</th>
<th>wtm</th>
<th>btm</th>
<th>DTC Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBK</td>
<td>0010.00</td>
<td>3 2-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNK</td>
<td>0001.00</td>
<td>3 2-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KPK</td>
<td>0006.10</td>
<td>3 2-1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQK</td>
<td>1000.00</td>
<td>3 2-1</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRK</td>
<td>0100.00</td>
<td>3 2-1</td>
<td>139</td>
<td>433</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBK</td>
<td>0040.00</td>
<td>4 2-2</td>
<td>52</td>
<td>14</td>
<td>14</td>
<td>52</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBN</td>
<td>0013.00</td>
<td>4 2-2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBP</td>
<td>0010.01</td>
<td>4 2-2</td>
<td>104</td>
<td>28</td>
<td>6</td>
<td>14</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>KBNK</td>
<td>0004.00</td>
<td>4 2-2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>KBNP</td>
<td>0001.01</td>
<td>4 2-2</td>
<td>29</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>KPKN</td>
<td>0000.11</td>
<td>4 2-2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>KQKB</td>
<td>1030.00</td>
<td>4 2-2</td>
<td>980</td>
<td>4,837</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKN</td>
<td>1003.00</td>
<td>4 2-2</td>
<td>5</td>
<td>19</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKP</td>
<td>1000.01</td>
<td>4 2-2</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>26</td>
<td>26</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>KQKK</td>
<td>1300.00</td>
<td>4 2-2</td>
<td>2</td>
<td>11</td>
<td>55</td>
<td>291</td>
<td>31</td>
<td>31</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>KRKB</td>
<td>0130.00</td>
<td>4 2-2</td>
<td>29</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRKN</td>
<td>0103.00</td>
<td>4 2-2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>27</td>
<td>27</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>KRKP</td>
<td>0100.01</td>
<td>4 2-2</td>
<td>28</td>
<td>42</td>
<td>3</td>
<td>3</td>
<td>16</td>
<td>16</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRRK</td>
<td>0400.00</td>
<td>4 2-2</td>
<td>59</td>
<td>111</td>
<td>111</td>
<td>59</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>KBBK</td>
<td>0200.00</td>
<td>4 3-1</td>
<td>16</td>
<td>59</td>
<td>0</td>
<td>0</td>
<td>19</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBNK</td>
<td>0011.00</td>
<td>4 3-1</td>
<td>144</td>
<td>436</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBP</td>
<td>0010.10</td>
<td>4 3-1</td>
<td>2</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBN</td>
<td>0002.00</td>
<td>4 3-1</td>
<td>77</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBNP</td>
<td>0001.10</td>
<td>4 3-1</td>
<td>24</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>22</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBN</td>
<td>0000.20</td>
<td>4 3-1</td>
<td>62</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKB</td>
<td>1010.00</td>
<td>4 3-1</td>
<td>2,411</td>
<td>14,012</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKN</td>
<td>1001.00</td>
<td>4 3-1</td>
<td>4,932</td>
<td>23,203</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKK</td>
<td>1000.10</td>
<td>4 3-1</td>
<td>75</td>
<td>175</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQQQ</td>
<td>2000.00</td>
<td>4 3-1</td>
<td>3,280</td>
<td>13,005</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQKR</td>
<td>1100.00</td>
<td>4 3-1</td>
<td>44</td>
<td>158</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBKB</td>
<td>0210.00</td>
<td>4 3-1</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBKN</td>
<td>0050.00</td>
<td>5 3-2</td>
<td>503</td>
<td>6</td>
<td>141</td>
<td>546</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>KBBKP</td>
<td>0023.00</td>
<td>5 3-2</td>
<td>34</td>
<td>53</td>
<td>44</td>
<td>222</td>
<td>66</td>
<td>66</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>KBBP</td>
<td>0020.01</td>
<td>5 3-2</td>
<td>34</td>
<td>69</td>
<td>5</td>
<td>11</td>
<td>21</td>
<td>21</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>KBBQ</td>
<td>3020.00</td>
<td>5 3-2</td>
<td>248</td>
<td>58</td>
<td>74</td>
<td>15</td>
<td>4</td>
<td>3</td>
<td>71</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>KBBR</td>
<td>0320.00</td>
<td>5 3-2</td>
<td>26</td>
<td>7</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>KBBK</td>
<td>0041.00</td>
<td>5 3-2</td>
<td>28</td>
<td>19</td>
<td>133</td>
<td>514</td>
<td>13</td>
<td>12</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>KBBN</td>
<td>0014.00</td>
<td>5 3-2</td>
<td>2</td>
<td>1</td>
<td>104</td>
<td>533</td>
<td>77</td>
<td>76</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>KBNP</td>
<td>0011.01</td>
<td>5 3-2</td>
<td>1</td>
<td>2</td>
<td>523</td>
<td>535</td>
<td>26</td>
<td>26</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>KBNK</td>
<td>0311.00</td>
<td>5 3-2</td>
<td>79</td>
<td>1</td>
<td>22</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>42</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>KBBR</td>
<td>0311.00</td>
<td>5 3-2</td>
<td>127</td>
<td>23</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>KBKP</td>
<td>0040.10</td>
<td>5 3-2</td>
<td>14</td>
<td>14</td>
<td>508</td>
<td>1,524</td>
<td>40</td>
<td>39</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>KBPN</td>
<td>0013.10</td>
<td>5 3-2</td>
<td>16</td>
<td>6</td>
<td>23</td>
<td>86</td>
<td>42</td>
<td>42</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>KBBP</td>
<td>0010.11</td>
<td>5 3-2</td>
<td>92</td>
<td>52</td>
<td>27</td>
<td>23</td>
<td>53</td>
<td>53</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>KBPK</td>
<td>3010.10</td>
<td>5 3-2</td>
<td>30</td>
<td>30</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>42</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>KBPR</td>
<td>0310.10</td>
<td>5 3-2</td>
<td>76</td>
<td>53</td>
<td>5</td>
<td>6</td>
<td>13</td>
<td>12</td>
<td>20</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1a.* Chess Endgames: 3-to-5-man DTC data.
Chess Endgames: Data and Strategy

<table>
<thead>
<tr>
<th>Endgame</th>
<th>GBR</th>
<th># w-b</th>
<th># of maximal positions</th>
<th>DTC Metric</th>
<th>max depths, moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>wtm  btm</td>
</tr>
<tr>
<td>KNNKB</td>
<td>0032.00</td>
<td>5 3-2</td>
<td>251 82 51 109</td>
<td>4 3 0 1</td>
<td>7 6 0 1</td>
</tr>
<tr>
<td>KNNKN</td>
<td>0005.00</td>
<td>5 3-2</td>
<td>38 18 56 293</td>
<td>114 113 12 13</td>
<td>1 0 63 63</td>
</tr>
<tr>
<td>KNNKP</td>
<td>0002.01</td>
<td>5 3-2</td>
<td>2 4 1 1</td>
<td>3 2 10 11</td>
<td></td>
</tr>
<tr>
<td>KNNQ</td>
<td>3002.00</td>
<td>5 3-2</td>
<td>2,387 465 10 2</td>
<td>31 30 8 9</td>
<td></td>
</tr>
<tr>
<td>KNNR</td>
<td>0302.00</td>
<td>5 3-2</td>
<td>2 1 6 11</td>
<td>1 0 63 63</td>
<td></td>
</tr>
<tr>
<td>KNPB</td>
<td>0031.10</td>
<td>5 3-2</td>
<td>11 3 5 18</td>
<td>33 33 13 14</td>
<td></td>
</tr>
<tr>
<td>KPNK</td>
<td>0004.10</td>
<td>5 3-2</td>
<td>9 2 27 132</td>
<td>5 4 43 43</td>
<td></td>
</tr>
<tr>
<td>KPKP</td>
<td>0001.11</td>
<td>5 3-2</td>
<td>1 6 6 9</td>
<td>18 18 42 43</td>
<td></td>
</tr>
<tr>
<td>KPKD</td>
<td>0000.21</td>
<td>5 3-2</td>
<td>3 11 66 58</td>
<td>30 29 13 14</td>
<td></td>
</tr>
<tr>
<td>KPPQ</td>
<td>3000.20</td>
<td>5 3-2</td>
<td>14 15 19 8</td>
<td>28 28 11 12</td>
<td></td>
</tr>
<tr>
<td>KPPDR</td>
<td>0300.20</td>
<td>5 3-2</td>
<td>1 1 2 3</td>
<td>5 6 30 30</td>
<td></td>
</tr>
<tr>
<td>KPBQ</td>
<td>1040.00</td>
<td>5 3-2</td>
<td>220 998 187 645</td>
<td>25 24 25 25</td>
<td></td>
</tr>
<tr>
<td>KPBQK</td>
<td>1013.00</td>
<td>5 3-2</td>
<td>74 343 30 153</td>
<td>9 9 0 1</td>
<td></td>
</tr>
<tr>
<td>KPBQK</td>
<td>1010.01</td>
<td>5 3-2</td>
<td>5 19 791 789</td>
<td>17 17 1 2</td>
<td></td>
</tr>
<tr>
<td>KPBQK</td>
<td>4010.00</td>
<td>5 3-2</td>
<td>33 1 1 1</td>
<td>35 35 13 14</td>
<td></td>
</tr>
<tr>
<td>KQBQ</td>
<td>1310.00</td>
<td>5 3-2</td>
<td>1 6 8,848 52,298</td>
<td>30 30 16 17</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1031.00</td>
<td>5 3-2</td>
<td>50 158 28 64</td>
<td>19 19 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1004.00</td>
<td>5 3-2</td>
<td>7 39 31 166</td>
<td>9 9 0 1</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1001.01</td>
<td>5 3-2</td>
<td>7 8 928 911</td>
<td>17 17 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>4001.00</td>
<td>5 3-2</td>
<td>7 1 1 4</td>
<td>35 35 13 14</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1301.00</td>
<td>5 3-2</td>
<td>1 6 15 86</td>
<td>22 22 2 3</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1030.10</td>
<td>5 3-2</td>
<td>1,122 4,328 374 1,290</td>
<td>9 9 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1010.01</td>
<td>5 3-2</td>
<td>1 6 3 9</td>
<td>10 10 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1010.01</td>
<td>5 3-2</td>
<td>11,817 39,633</td>
<td>6 6 3 3</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>4000.10</td>
<td>5 3-2</td>
<td>5 13 2 4</td>
<td>114 113 15 16</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1300.10</td>
<td>5 3-2</td>
<td>4 20 5,177 26,128</td>
<td>20 20 2 3</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>2030.00</td>
<td>5 3-2</td>
<td>4 15 0 0</td>
<td>4 4 — —</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>2003.00</td>
<td>5 3-2</td>
<td>287 1,411 0 0</td>
<td>4 4 — —</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>2000.01</td>
<td>5 3-2</td>
<td>18,995 19,257 140 140</td>
<td>3 3 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>5000.00</td>
<td>5 3-2</td>
<td>2 21 31 152</td>
<td>25 25 6 7</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>2300.00</td>
<td>5 3-2</td>
<td>2 12 2,383 16,681</td>
<td>14 14 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1130.00</td>
<td>5 3-2</td>
<td>720 2,556 0 0</td>
<td>5 5 — —</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1103.00</td>
<td>5 3-2</td>
<td>234 1,149 36 149</td>
<td>5 5 0 1</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1100.01</td>
<td>5 3-2</td>
<td>104,508 131,846 683 683</td>
<td>3 3 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>4100.00</td>
<td>5 3-2</td>
<td>3 31 1 2</td>
<td>60 60 8 9</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>1400.00</td>
<td>5 3-2</td>
<td>10 54 8,099 56,501</td>
<td>15 15 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0140.00</td>
<td>5 3-2</td>
<td>35 46 251 951</td>
<td>25 25 1 2</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0113.00</td>
<td>5 3-2</td>
<td>9 35 106 481</td>
<td>21 21 0 1</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0110.01</td>
<td>5 3-2</td>
<td>2 12 4 12</td>
<td>11 11 4 5</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>3110.00</td>
<td>5 3-2</td>
<td>1 3 5 4</td>
<td>7 6 41 42</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0410.00</td>
<td>5 3-2</td>
<td>28 19 3 14</td>
<td>59 58 3 4</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0131.00</td>
<td>5 3-2</td>
<td>3 6 41 89</td>
<td>25 25 0 1</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0104.00</td>
<td>5 3-2</td>
<td>5 18 101 468</td>
<td>24 24 0 1</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0101.01</td>
<td>5 3-2</td>
<td>65 81 2 2</td>
<td>15 15 10 11</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>3101.00</td>
<td>5 3-2</td>
<td>24 5 7 3</td>
<td>9 8 46 46</td>
<td></td>
</tr>
<tr>
<td>KQNB</td>
<td>0401.00</td>
<td>5 3-2</td>
<td>1 1 1 3</td>
<td>33 32 4 5</td>
<td></td>
</tr>
</tbody>
</table>

Table 1b. Chess Endgames: 3-to-5-man DTC data.
Table le. Chess Endgames: 3-to-5-man DTC data.

<table>
<thead>
<tr>
<th>Endgame</th>
<th>Name</th>
<th>GBR</th>
<th># w-b</th>
<th>wtm</th>
<th>btm</th>
<th>wtm</th>
<th>btm</th>
<th>max depths, moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1-0</td>
<td>0-1</td>
<td>1-0</td>
<td>0-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 130.10  | KRPKB   | 5   | 3-2   | 11  | 26  | 502 | 1,672 | 62 | 1 | 2 |
| 0103.10 | KRPKN   | 5   | 3-2   | 2   | 7   | 4   | 12   | 46 | 1 | 2 |
| 0100.11 | KRPKP   | 5   | 3-2   | 184 | 474 | 17  | 17   | 9  | 10 | 11 |
| 3100.10 | KRPKQ   | 5   | 3-2   | 5   | 5   | 5   | 1    | 9  | 8  | 79 |
| 0400.10 | KRPKR   | 5   | 3-2   | 33  | 4   | 23  | 80   | 60 | 6  | 7 |
| 0230.00 | KRRKB   | 5   | 3-2   | 1   | 4   | 0   | 0    | 10 | 10 | — |
| 0203.00 | KRRKN   | 5   | 3-2   | 215 | 687 | 45  | 184  | 7  | 7  | 0  |
| 0200.01 | KRRKP   | 5   | 3-2   | 16  | 48  | 988 | 988  | 9  | 9  | 1  |
| 3200.00 | KRRQK   | 5   | 3-2   | 14  | 4   | 2   | 3    | 15 | 14 | 20 | 20 |
| 0500.00 | KRRKR   | 5   | 3-2   | 3   | 15  | 6,210| 43,225 | 25 | 25 | 1 | 2 |
| 0090.00 | KBBKB   | 5   | 4-1   | 116 | 345 | 0   | 0    | 10 | 10 | — |
| 0021.00 | KBBNK   | 5   | 4-1   | 783 | 2,066| 0   | 0    | 13 | 13 | — |
| 0020.10 | KBBPK   | 5   | 4-1   | 3   | 2   | 0   | 0    | 16 | 16 | — |
| 0012.00 | KBNPK   | 5   | 4-1   | 22  | 59  | 0   | 0    | 13 | 13 | — |
| 0011.10 | KBNPQ   | 5   | 4-1   | 9   | 45  | 0   | 0    | 10 | 10 | — |
| 0010.20 | KBPPQ   | 5   | 4-1   | 56  | 46  | 0   | 0    | 16 | 16 | — |
| 0009.00 | KNNNK   | 5   | 4-1   | 44  | 180 | 0   | 0    | 21 | 21 | — |
| 0002.10 | KNNPK   | 5   | 4-1   | 194 | 296 | 0   | 0    | 15 | 15 | — |
| 0001.20 | KNPQK   | 5   | 4-1   | 2   | 5   | 0   | 0    | 12 | 12 | — |
| 0000.30 | KPPQK   | 5   | 4-1   | 11  | 35  | 0   | 0    | 11 | 11 | — |
| 1020.00 | QQBBK   | 5   | 4-1   | 182 | 673 | 0   | 0    | 6  | 6  | — |
| 1011.00 | QQBNK   | 5   | 4-1   | 54,680| 236,453| 0   | 0   | 4  | 4  | — |
| 1010.10 | QBBPK   | 5   | 4-1   | 68  | 255 | 0   | 0    | 6  | 6  | — |
| 1002.00 | KQQNK   | 5   | 4-1   | 11,789| 56,328| 0   | 0   | 5  | 5  | — |
| 1001.10 | KNNPK   | 5   | 4-1   | 1,264| 4,476| 0   | 0    | 6  | 6  | — |
| 1000.20 | KQBBK   | 5   | 4-1   | 96,576| 412,131| 0   | 0   | 3  | 3  | — |
| 2010.00 | KQBNK   | 5   | 4-1   | 13  | 58  | 0   | 0    | 4  | 4  | — |
| 2001.00 | KQBPK   | 5   | 4-1   | 138 | 732 | 0   | 0    | 4  | 4  | — |
| 2000.10 | QBBPK   | 5   | 4-1   | 1,513| 6,553| 0   | 0    | 3  | 3  | — |
| 9000.00 | KQBBK   | 5   | 4-1   | 56,174| 218,959| 0   | 0   | 3  | 3  | — |
| 2100.00 | KQRBK   | 5   | 4-1   | 1,198| 5,865| 0   | 0    | 4  | 4  | — |
| 1100.00 | KQKPQ   | 5   | 4-1   | 7,474| 31,526| 0   | 0   | 4  | 4  | — |
| 1100.10 | KQRPK   | 5   | 4-1   | 3   | 15  | 0   | 0    | 5  | 5  | — |
| 1200.00 | KQQNK   | 5   | 4-1   | 18  | 87  | 0   | 0    | 4  | 4  | — |
| 0120.00 | KRRBK   | 5   | 4-1   | 24  | 126 | 0   | 0    | 10 | 10 | — |
| 0110.00 | KRRKN   | 5   | 4-1   | 8,391| 26,677| 0   | 0   | 7  | 7  | — |
| 0102.00 | KRRPK   | 5   | 4-1   | 1   | 5   | 0   | 0    | 8  | 8  | — |
| 0100.10 | KRRQK   | 5   | 4-1   | 602 | 2,052| 0   | 0    | 10 | 10 | — |
| 0010.20 | KRRPPK  | 5   | 4-1   | 579 | 1,436| 0   | 0    | 8  | 8  | — |
| 0010.20 | KRRPK   | 5   | 4-1   | 4   | 24  | 0   | 0    | 8  | 8  | — |
| 0210.00 | KRRBK   | 5   | 4-1   | 4,761| 17,210| 0   | 0   | 5  | 5  | — |
| 0201.00 | KRRNK   | 5   | 4-1   | 8,533| 29,009| 0   | 0   | 5  | 5  | — |
| 0200.10 | KRRPK   | 5   | 4-1   | 16  | 56  | 0   | 0    | 6  | 6  | — |
| 0900.00 | KRRRK   | 5   | 4-1   | 3,566| 13,290| 0   | 0   | 4  | 4  | — |
# Chess Endgames: Data and Strategy

## DTZ Metric

<table>
<thead>
<tr>
<th>Endgame</th>
<th>GBR</th>
<th># w-b</th>
<th># of maximal positions</th>
<th>DTZ Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-0</td>
<td>0-1</td>
</tr>
<tr>
<td>KPK</td>
<td>0000.10</td>
<td>3</td>
<td>2-1</td>
<td>8</td>
</tr>
<tr>
<td>KBBK</td>
<td>0020.01</td>
<td>5</td>
<td>3-2</td>
<td>16</td>
</tr>
<tr>
<td>KBNKN</td>
<td>0011.05</td>
<td>5</td>
<td>3-2</td>
<td>202</td>
</tr>
<tr>
<td>KPBK</td>
<td>0000.20</td>
<td>4</td>
<td>3-1</td>
<td>125</td>
</tr>
<tr>
<td>KQPK</td>
<td>1000.10</td>
<td>4</td>
<td>3-1</td>
<td>25</td>
</tr>
<tr>
<td>KPPK</td>
<td>0001.11</td>
<td>5</td>
<td>3-2</td>
<td>89</td>
</tr>
<tr>
<td>KBBKQ</td>
<td>3001.10</td>
<td>5</td>
<td>3-2</td>
<td>1,438</td>
</tr>
<tr>
<td>KBBKR</td>
<td>0301.10</td>
<td>5</td>
<td>3-2</td>
<td>5</td>
</tr>
<tr>
<td>KNPKN</td>
<td>0004.10</td>
<td>5</td>
<td>3-2</td>
<td>2</td>
</tr>
<tr>
<td>KNPKQ</td>
<td>0001.11</td>
<td>5</td>
<td>3-2</td>
<td>1</td>
</tr>
<tr>
<td>KBBKQ</td>
<td>3001.10</td>
<td>5</td>
<td>3-2</td>
<td>2,459</td>
</tr>
<tr>
<td>KBBKR</td>
<td>0301.10</td>
<td>5</td>
<td>3-2</td>
<td>8</td>
</tr>
<tr>
<td>KNKP</td>
<td>0002.01</td>
<td>5</td>
<td>3-2</td>
<td>18</td>
</tr>
<tr>
<td>KNPKB</td>
<td>0031.10</td>
<td>5</td>
<td>3-2</td>
<td>39</td>
</tr>
<tr>
<td>KNPK</td>
<td>0004.10</td>
<td>5</td>
<td>3-2</td>
<td>2</td>
</tr>
<tr>
<td>KPQK</td>
<td>4000.10</td>
<td>5</td>
<td>3-2</td>
<td>1</td>
</tr>
<tr>
<td>KQPK</td>
<td>1300.10</td>
<td>5</td>
<td>3-2</td>
<td>6,992</td>
</tr>
</tbody>
</table>

Table 2a. Chess Endgames: 3-to-5-man DTZ data.
Table 2b. Chess Endgames: 3-to-5-man DTZ data.

<table>
<thead>
<tr>
<th>Endgame</th>
<th>GBR</th>
<th># w-b</th>
<th># of maximal positions</th>
<th>max depth, moves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>1-0</td>
<td>0-1</td>
</tr>
<tr>
<td></td>
<td>wtm</td>
<td>btm</td>
<td>wtm</td>
<td>btm</td>
</tr>
<tr>
<td>KBBKN</td>
<td>0023.00</td>
<td>5 3-2</td>
<td>347,796</td>
<td>485,538</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>222</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBKP</td>
<td>0020.01</td>
<td>5 3-2</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBBKQ</td>
<td>3020.00</td>
<td>5 3-2</td>
<td>248</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>86,896</td>
<td>24,793</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBNKQ</td>
<td>0014.00</td>
<td>5 3-2</td>
<td>12,123</td>
<td>5,857</td>
</tr>
<tr>
<td></td>
<td>104</td>
<td>533</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBNKP</td>
<td>0011.01</td>
<td>5 3-2</td>
<td>202</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>494</td>
<td>157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KBPKN</td>
<td>0013.10</td>
<td>5 3-2</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>86</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNNKP</td>
<td>0002.01</td>
<td>5 3-2</td>
<td>60,080</td>
<td>12,023</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNNKQ</td>
<td>3002.00</td>
<td>5 3-2</td>
<td>2,387</td>
<td>465</td>
</tr>
<tr>
<td></td>
<td>6,352</td>
<td>2,010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNNKN</td>
<td>0004.11</td>
<td>5 3-2</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>132</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KNPKN</td>
<td>3001.10</td>
<td>5 3-2</td>
<td>2,459</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KPPKP</td>
<td>0000.21</td>
<td>5 3-2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KPKPQ</td>
<td>3000.20</td>
<td>5 3-2</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQQKP</td>
<td>1000.11</td>
<td>5 3-2</td>
<td>69</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1,024</td>
<td>7,412,631</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KQPQK</td>
<td>4000.10</td>
<td>5 3-2</td>
<td>1,595</td>
<td>2,415</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRRKP</td>
<td>1100.01</td>
<td>5 3-2</td>
<td>76,181</td>
<td>2,592</td>
</tr>
<tr>
<td></td>
<td>683</td>
<td>892,287</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRRKQ</td>
<td>4100.00</td>
<td>5 3-2</td>
<td>23</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRBKR</td>
<td>0410.01</td>
<td>5 3-2</td>
<td>1,041</td>
<td>175</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRPKB</td>
<td>0130.10</td>
<td>5 3-2</td>
<td>130</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td>502</td>
<td>1,672</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRPKF</td>
<td>0100.11</td>
<td>5 3-2</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KRPQK</td>
<td>3100.10</td>
<td>5 3-2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>9,275</td>
<td>4,898</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Chess Endgames: 3-to-5-man data where EZ<sub>50</sub> ≠ EZ.
Chess Endgames: Data and Strategy

### Table 4. Chess Endgames: some 6-man DTZ data.

<table>
<thead>
<tr>
<th>Endgame</th>
<th>GBR</th>
<th># w-b</th>
<th>1-0 wtm</th>
<th>1-0 btm</th>
<th>0-1 wtm</th>
<th>0-1 btm</th>
<th>max depth, moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBBKNN</td>
<td>0026.00</td>
<td>6 3-3</td>
<td>11</td>
<td>1</td>
<td>488</td>
<td>1,518</td>
<td>38 38 3 4</td>
</tr>
<tr>
<td>KQQKBB</td>
<td>2060.00</td>
<td>6 3-3</td>
<td>984</td>
<td>5,128</td>
<td>137</td>
<td>714</td>
<td>6 6 3 4</td>
</tr>
<tr>
<td>KQQKKN</td>
<td>2006.00</td>
<td>6 3-3</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>36,110</td>
<td>7 7 1 1</td>
</tr>
<tr>
<td>KQQQKR</td>
<td>5300.00</td>
<td>6 4-2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>12</td>
<td>48 47 56 56</td>
</tr>
<tr>
<td>KRRKRB</td>
<td>0530.00</td>
<td>6 3-3</td>
<td>22</td>
<td>13</td>
<td>1</td>
<td>455</td>
<td>54 54 6 6</td>
</tr>
<tr>
<td>KBBKKN</td>
<td>0093.00</td>
<td>6 4-2</td>
<td>6</td>
<td>6</td>
<td>951</td>
<td>4,838</td>
<td>12 12 0 1</td>
</tr>
<tr>
<td>KBBKQ</td>
<td>3090.00</td>
<td>6 4-2</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>10 9 51 51</td>
</tr>
<tr>
<td>KBBKNN</td>
<td>0024.00</td>
<td>6 4-2</td>
<td>9</td>
<td>54</td>
<td>3,663</td>
<td>18,984</td>
<td>31 31 0 1</td>
</tr>
<tr>
<td>KBBNNK</td>
<td>0015.00</td>
<td>6 4-2</td>
<td>17</td>
<td>56</td>
<td>4,335</td>
<td>22,890</td>
<td>28 28 0 1</td>
</tr>
<tr>
<td>KBBNNQ</td>
<td>3012.00</td>
<td>6 4-2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>12 11 49 49</td>
</tr>
<tr>
<td>KNNNQK</td>
<td>3009.00</td>
<td>6 4-2</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>9 8 35 35</td>
</tr>
<tr>
<td>KQNNQK</td>
<td>4002.00</td>
<td>6 4-2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>20</td>
<td>71 71 13 14</td>
</tr>
<tr>
<td>KRNKQ</td>
<td>3102.00</td>
<td>6 4-2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>28 27 41 41</td>
</tr>
</tbody>
</table>

### Table 5. Chess Endgames: some 6-man DTZ50 data.

<table>
<thead>
<tr>
<th>Endgame</th>
<th>GBR</th>
<th># w-b</th>
<th>1-0 wtm</th>
<th>1-0 btm</th>
<th>0-1 wtm</th>
<th>0-1 btm</th>
<th>max depth, moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBBKNN</td>
<td>0026.00</td>
<td>6 3-3</td>
<td>46</td>
<td>17</td>
<td>488</td>
<td>1,518</td>
<td>29 28 3 4</td>
</tr>
<tr>
<td>KQQKBB</td>
<td>2060.00</td>
<td>6 3-3</td>
<td>1</td>
<td>5</td>
<td>137</td>
<td>714</td>
<td>8 8 3 4</td>
</tr>
<tr>
<td>KQQKKN</td>
<td>2006.00</td>
<td>6 3-3</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>36,110</td>
<td>7 7 1 1</td>
</tr>
<tr>
<td>KQQQKR</td>
<td>5300.00</td>
<td>6 4-2</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>26</td>
<td>48 47 50 50</td>
</tr>
<tr>
<td>KRRKRB</td>
<td>0530.00</td>
<td>6 3-3</td>
<td>372</td>
<td>107</td>
<td>1</td>
<td>455</td>
<td>50 50 6 6</td>
</tr>
<tr>
<td>KBBKKN</td>
<td>0093.00</td>
<td>6 4-2</td>
<td>3</td>
<td>6</td>
<td>951</td>
<td>4,838</td>
<td>14 14 0 1</td>
</tr>
<tr>
<td>KBBKQ</td>
<td>3090.00</td>
<td>6 4-2</td>
<td>1</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>10 9 50 50</td>
</tr>
<tr>
<td>KBBKNN</td>
<td>0024.00</td>
<td>6 4-2</td>
<td>9</td>
<td>54</td>
<td>3,663</td>
<td>18,984</td>
<td>31 31 0 1</td>
</tr>
<tr>
<td>KBBNNK</td>
<td>0015.00</td>
<td>6 4-2</td>
<td>3</td>
<td>3</td>
<td>4,335</td>
<td>22,890</td>
<td>29 29 0 1</td>
</tr>
<tr>
<td>KBBNNQ</td>
<td>3012.00</td>
<td>6 4-2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>12 11 49 49</td>
</tr>
<tr>
<td>KNNNQK</td>
<td>3009.00</td>
<td>6 4-2</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>11</td>
<td>9 8 35 35</td>
</tr>
<tr>
<td>KQNNQK</td>
<td>4002.00</td>
<td>6 4-2</td>
<td>10,534</td>
<td>9,796</td>
<td>5</td>
<td>20</td>
<td>50 50 13 14</td>
</tr>
<tr>
<td>KRNKQ</td>
<td>3102.00</td>
<td>6 4-2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>28 27 41 41</td>
</tr>
</tbody>
</table>

The following lines, starting from some positions listed in Table 7 below, show strategies variously retaining the win, failing to retain the win, repeating positions to draw or being suboptimal. They include an established notation showing the criticality of the moves:

" = unique value-preserving move; ' = only optimal move; o = only legal move.

KBBK position BB-P1 − dtz = 1m; dtz50 = 7m:
Sφ − Szφ, σ = C, M or Z: 1 ... a1Q+?? \{dtz = 51m; White can force a 50m draw\} ½-½.

SZ50+ − SZ50: 1 ... Kg4" 2. Bf3+ Ke3 3. Be1+’ Nd4" 4. Bf2+’ Ke5 5. Bg3+’ Kf6’ 6. Bh4+’
Kg7" \{dtm = 17m\} 0-1.

KNNKP position NN-P1 − dtz = 20m, dtc = 63m, dtm = 64m, dtz50 = 44m:
KNNKP position NN-P2 – $dtz = 1m$, $dtz_{50} = 43m$:

$SZ\sigma - SzT$: 1. Nb6's?? \{ $dtz = 58m$; Black can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.

S(CM)/$\sigma - SzT_{50}$: 1. Na4# \{ $dtz_{50} = 42m$, $dtm = 88m$ \} Kd2' 1-0.


KNNKP position NN-P3 – $dtz = 1m$, $dtz_{50} = 31m$:

$SZ\sigma - SzT$, $\sigma = C'$, $M'$, $Z'$ or $Z_{50}$: 1. Ke2? \{ $dtz > 51m$. \}

$SZ'_{50} - SH_{50}$: 1. Nb6'+ \{ $dtz = 51m$, controlling DTR \} Ka4'

KNNKP position NN-P4 – $dtz = 16m$, $dtz_{50} = 25m$, $mleft = 25m$:

$SZ'_{50} - SH_{50} +$: 1. Nd5+? \{ $dtz_{50} = 26m$ \} Ke4' 2. Ndc3 Kb4' \{ NN-P4 repeated \} $\frac{1}{2}$-$\frac{1}{2}$.

KQPQK position QP-Q1 – $dtz = 52m$, $dtz = 1m$, $dtz_{50} = 50m$:

$S\sigma - SzT$, $\sigma = C'$, $M'$ or $Z'$: 1. b7'?? \{ $dtz = 51m$; Black can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.


KRKPQ position RP-P2 – $dtz = 1m$, $dtz_{50} = 6m$:

$S\phi - SzT$, $\phi = C'$, $M'$ or $Z'$: 1. ... QI'?? \{ $dtz > 50m$; White can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.

$SZ_{50} - SZ_{50}$: 1. ... Kb2' 2. Rb4' Kc2' 3. Rc4'+ Kd2' 4. Rd4'+ Ke2' 5. Re4'+ Kf2' 6. Re7 Qg1' \{ $dtm = 49m$ \} 0-1.

KRKPQ position RP-Q1 – $dtz = 2m$, $dtz_{50} = 21m$:

$S\phi - SzT$, $\phi = C'$ or $M'$: 1. ... Qd6'?? \{ $dtz > 50m$; White can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.

$SZ_{50} - SZ_{50}$: 1. ... Qe4'?? \{ $dtz > 50m$; White can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.

KKBKNN position BB-NN1 – $dtz = 1m$, $dtz_{50} = 28m$:

$S\sigma - SzT$, $\sigma = C'$, $M'$ or $Z'$: 1. Bxg6'?? \{ $dtz = 54m$; Black can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.


KQQKBB position QQ-BB1 – $dtz = 2m$, $dtz_{50} = 7m$:

$SZ' - SzF$: 1. Qxd4'?? Bxd4' \{ $dtz = 67m$; Black can force a 50m draw \} $\frac{1}{2}$-$\frac{1}{2}$.

Chess Endgames: Data and Strategy

KBNNKQ position BNN-Q1 - dtz = 1m, dtz50 = 36m:

\[ \text{S}^f \sim \text{S}^g, \sigma = C', M' \lor Z': 1. \ldots \text{Qxa1}?? \ (dtz = 52m; \text{Black can force a 50m draw}) \frac{1}{2}-\frac{1}{2}. \]

\[ \text{SZ}_50 \sim \text{SZ}_90; 1. \ldots \text{Qh7}+^2. \text{Kd2} \text{Qd7}+^3. \text{Kc3} \text{Ke2}^4. \text{Bb2} \text{Qg4}^5. \text{Kb3} \text{Qe6}^6. \text{Kc3} \text{Qe4}^7. \text{Kb3} \text{Qg4}^8. \text{Kc3} \text{Qf4}^9. \text{Kb3} \text{Qb4}^10. \text{Kc2} \text{Qb4}^11. \text{Na3} \text{Qe4}^12. \text{Kb3} \text{Qd5}^+13. \text{Kc3} \text{Qf5}^+14. \text{Kc4} \text{Kd1} 15. \text{Kb4} \text{Qb7}^+16. \text{Nb5} \text{Kc2}^17. \text{Bd4} \text{Qe7}^+18. \text{Kc4} \text{Qe6}^+19. \text{Kc5} \text{Qf5}^+20. \text{Kc4} \text{Qe8}^+21. \text{Kb4} \text{Qf8}^+22. \text{Ka4} \text{Qg8}^23. \text{Kb4} \text{Kd3}^24. \text{Bc3} \text{Qd5}^25. \text{Bd4} \text{Qc4}^+26. \text{Ka5} \text{Qg8}^27. \text{Ka4} \text{Qa8}^+28. \text{Kb4} \text{Qf8}^+29. \text{Kb3} \text{Qe7}^30. \text{Bb2} \text{Qe6}^+31. \text{Ka4} \text{Qa2}^32. \text{Ba3} \text{Qc4}^+33. \text{Ka5} \text{Qe4}^+34. \text{Kb4} \text{Qe6}^+35. \text{Ka5} \text{Qa8}^+36. \text{Kb6} \text{Qxh8} \{\text{dtm} = 22m\} 0-1. \]

KQNNKQ position QNN-Q1 - dtz = 3m, dtz50 = 4m, dtm = 5m:

\[ \text{SZ}_50 \sim \text{SZ}_90; 1. \ldots \text{Qa3}+?? \text{Kd1}^2. \text{Qa1}^+3. \text{Kc2}^3. \text{Qxh1}^4 \{\text{dtz = 52m\} \frac{1}{2}-\frac{1}{2}. \]

---

### Table 6. The impact of the 50-move drawing rule.

<table>
<thead>
<tr>
<th>Endgame</th>
<th># extra draws</th>
<th>% of nominal wins</th>
<th># delayed</th>
<th>% of nominal wins</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>extra draws</td>
<td></td>
<td>delayed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wtm</td>
<td>btm</td>
<td>wtm</td>
<td>btm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>wtm</td>
<td>btm</td>
</tr>
<tr>
<td>KBBKN</td>
<td>1-0</td>
<td>3,993,656</td>
<td>7,852,543</td>
<td>7,852,543</td>
</tr>
<tr>
<td>KBBKP</td>
<td>1-0</td>
<td>171</td>
<td>3,889</td>
<td>1,800</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>119,226</td>
<td>1,524</td>
<td>3,741</td>
</tr>
<tr>
<td>KBBKQ</td>
<td>0-1</td>
<td>2,154,114</td>
<td>490,797</td>
<td>490,797</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>213</td>
<td>1,641</td>
<td>1,685</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>257</td>
<td>602</td>
<td>1,530</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>10,684,968</td>
<td>17,093,973</td>
<td>17,093,973</td>
</tr>
<tr>
<td>KNNQ</td>
<td>0-1</td>
<td>4,255</td>
<td>301</td>
<td>357</td>
</tr>
<tr>
<td>KNPKN</td>
<td>0-1</td>
<td>11,990</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>61</td>
<td>48</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>1,834</td>
<td>149</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>1,641</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>19</td>
<td>2,664</td>
<td>2,207</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>28,468</td>
<td>42,756</td>
<td>28,526</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>230</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>2,263</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>35</td>
<td>53</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>679</td>
<td>26</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>1,592</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>72,802</td>
<td>29,723</td>
<td>29,723</td>
</tr>
<tr>
<td>KBBKN</td>
<td>1-0</td>
<td>141,874,223</td>
<td>38,562,549</td>
<td>38,562,549</td>
</tr>
<tr>
<td>KQQK</td>
<td>0-1</td>
<td>23,343</td>
<td>6,776,509</td>
<td>6,776,509</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>130</td>
<td>44,687</td>
<td>44,687</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>17,313</td>
<td>41,775</td>
<td>41,775</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>38</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>19</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>592</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>3,226</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 7. Example Positions.7

7 Without a DTR EGT, it is not always possible to determine dttr precisely.
EVALUATION IN GO BY A NEURAL NETWORK USING SOFT SEGMENTATION

M. Enzenberger
University of Alberta, Edmonton, Alberta, Canada
emarkus@cs.ualberta.ca, http://www.cs.ualberta.ca/~emarkus/

Abstract In this article a neural network architecture is presented that is able to build a soft segmentation of a two-dimensional input. This network architecture is applied to position evaluation in the game of Go. It is trained using self-play and temporal difference learning combined with a rich two-dimensional reinforcement signal. Two experiments are performed, one using the raw board position as input, the other one doing some simple preprocessing of the board. The second network is able to achieve playing strength comparable to a 13-kyu Go program.

Keywords: Go, neural networks, segmentation, connectivity, NeuroGo

1. Evaluating Go Positions

Writing a program that plays the game of Go is a notoriously hard problem. Despite many efforts the best programs still play at a weak to medium amateur level of about 8 kyu (Schaeffer, 2001). This is not only due to the large branching factor but also to the fact that the evaluation of Go positions is difficult. State-of-the-art programs rely on a knowledge intensive approach. They use large databases of patterns, rule-based systems, and hand-tuned heuristics (Bouzy and Cazenave, 2001).

1.1 Simplification by Segmentation

The evaluation of a Go position can be simplified by segmenting the position into parts. This works well in positions with independent subgames where playing one subgame does not affect the value of other subgames. A typical example are Go endgame positions to which combinatorial game theory has been applied successfully (Müller and Gasser, 1996). Positions without a clear segmentation are more difficult: in particular, middle-game positions with many possible continuations each leading to different follow-up segmentations, and positions with multiple nearby tactical fights. Many Go programs use some influence-based segmentation of positions (Chen, 2002).
Cognitive studies on human Go players have shown that humans perceive Go positions not as a set of hierarchical structured patterns with clear boundaries but rather as a set of overlapping clusters (Reitman, 1976).

1.2 Neural Networks

Neural networks have been used for evaluating full-board Go positions. Schraudolph, Dayan, and Sejnowski (1994) used temporal-difference learning (Sutton, 1988) to train a neural network to evaluate Go positions. They showed that it is important to use a rich reinforcement signal and a sparsely connected network architecture that reflects the local character and translational invariance of the pattern-recognition task. This was accomplished by using $5 \times 5$ receptive fields with weight sharing.

However, essential features of a Go position depend on whether two points on the board are connected by one colour or will become connected later. Using fixed-size receptive fields makes the recognition of long distance connections impossible. The most basic cases are blocks. Blocks are sets of adjacent stones of the same colour; they can take an arbitrary shape on the board. Moreover, they can only be captured as a unit.

The Go program NEUROGO (version 2) used receptive fields that dynamically adapt their size to fit around blocks (Enzenberger, 1996). This was achieved by transforming the Go position into a graph with all stones of a block merged into a single node. While NEUROGO’s performance was improved greatly compared to a network using fixed-size receptive fields, it was still impossible for the network to represent higher-level objects like groups. Groups are a set of loosely connected blocks that might become connected later. This was the motivation for the development of a new network architecture with better abilities for segmenting the board.

2. Architecture using Soft Segmentation

This section presents a neural-network architecture that is able to process a Go position by building a soft segmentation of the position. This architecture is now used in version 3 of NEUROGO.

The neural network uses a feedforward backpropagation architecture. The neurons have a sigmoid activation function, with activation values between 0 and 1 and a bias weight. The soft segmentation of a position is represented as two connectivity maps, one for each colour. Each connectivity map assigns a connectivity strength between 0 and 1 to each pair of points on the board. See Figure 1 for an overview of the network architecture.

The next section describes the reinforcement signal that is used for learning, followed by a description of the layers in the network and the connections between them.
2.1 Reinforcement Signal

The final position of a Go game contains much richer information than merely the global score. The network uses single-point eyes, connections, and live points which are defined as follows.

- A **single-point eye** is an empty point with all adjacent points occupied by stones of the same block or by stones of two blocks of the same colour that share another single-point eye.
- A pair of points is **connected** by one colour if there is a path between them containing only stones of that colour or single-point eyes.
- A point is said to be **alive** if it is connected to two single-point eyes.

Chinese scoring rules are used during the network training. No pass move is allowed until all points on the board are alive\(^1\). Also, it is not allowed to play in one’s own single-point eyes.

Single-point eyes and connections can occur in earlier positions of the game, but may not exist in the end position, because those blocks could have been captured. Live points stay alive from the first position in which they occur until the end position.

The network uses single-point eyes, connections and live points as a reinforcement signal. Connections are used only locally within a 3 × 3 window centred around each point.

---

\(^1\)This makes scoring and detection of the end of the game easy, but will lead to wrong play in case of seki situations or more complicated single-point eyes (involving more than two blocks). However these cases rarely occur in actual games.
2.2 Neuron Layers

Each layer of neurons in the architecture contains one or more neurons for each point on the board. There are 7 layers as follows.

Input layer: This layer contains one or more neurons per point depending on the number of (boolean) input features that are used. The activation of the neurons is set to 0 or 1 according to whether a certain input feature is present in the Go position at this point. A Go position is always transformed such that Black is to move. This makes an additional input for indicating what colour is to move unnecessary.

First hidden layer: This layer contains one or more neurons per point. The number of neurons per point is a parameter of the network architecture. The layer is connected with receptive fields to the input layer.

Second hidden layer: Like the first hidden layer, this layer contains one or more neurons per point. The number of neurons per point is another parameter of the network architecture. The layer is connected with receptive fields to the first hidden layer.

Simple eyes layer: This layer contains 2 neurons per point, one for each colour. The activation is a prediction of whether that colour is able to create a single-point eye at this point. The layer is connected with receptive fields to the first hidden layer. It receives a reinforcement signal when a simple eye is created on the board.

Local connections layer: This layer contains 18 neurons per point, 9 for each colour. The activation is a prediction of whether that colour is able to create a connection from this point to each of the 9 points in a 3 x 3 window around this point (including self-connection). Neurons corresponding to off-board points are unused. The layer is connected with receptive fields to the first hidden layer. It receives a reinforcement signal when a connection is created on the board.

Global connectivity layer: This layer contains 2 \cdot n^2 neurons per point for board size $n$. The activation is a prediction whether each colour is able to create a connection from this point to any point on the board. The activation is computed by the connectivity pathfinder (see 2.5) from the local connections layer.

Evaluation layer: This layer contains 1 neuron per point. The activation is a prediction whether this point will be alive for Black (activation 1) or White (activation 0). The layer is connected to the second hidden layer and the simple eyes layer by connectivity-based weight selection (see 2.6). It receives a reinforcement signal for live points when they are created on the board.
2.3  Point Types

Each neuron corresponds to a point. Different point types are defined. The actual weights are chosen from weight sets depending on the point type.

There are two reasons for using weight sets. They increase the number of free parameters without significantly affecting the time for processing a position, since only one weight of a set is selected. They also compensate for effects of the edge of the board while still making it possible to learn local patterns that are mostly invariant with respect to translation.

The function type(p) assigns a type to each point p. See Figure 2 for the point types that were used.

<table>
<thead>
<tr>
<th>type(p)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Empty corner point</td>
</tr>
<tr>
<td>1</td>
<td>Empty edge point next to corner</td>
</tr>
<tr>
<td>2</td>
<td>Other empty edge point</td>
</tr>
<tr>
<td>3</td>
<td>Empty point diagonal from corner</td>
</tr>
<tr>
<td>4</td>
<td>Other empty point on second line</td>
</tr>
<tr>
<td>5</td>
<td>Empty point on line 3 or higher</td>
</tr>
<tr>
<td>6</td>
<td>Black stone</td>
</tr>
<tr>
<td>7</td>
<td>White stone</td>
</tr>
</tbody>
</table>

Figure 2.  Point types: definition and example.

2.4  Receptive Fields

The function window(p) assigns to each point p the set of points within a 3 x 3 square window centred at this point. If a layer is connected with receptive fields to a previous layer then each neuron corresponding to a point p is connected to all neurons in the previous layer corresponding to the points p' ∈ window(p). The spatial relationship of two points p and p' is described by a field index given by the function field(p,p') (see Figure 3).

Consider a layer L with n neurons per point connected to a previous layer L' with m neurons per point by receptive fields. Then a neuron corresponding to a point p and index i ∈ {1..n} is connected to all neurons in the previous layer corresponding to points p' ∈ window(p) and index j ∈ {1..m} using the weights

\[ w_{i,j}^{LL'} \]

\[ u_{i,j}^{L,L'}(\text{type}(p),\text{type}(p'),\text{field}(p,p')) \]

The neuron has a bias weight \[ b_i^L \text{type}(p) \].
2.5 Connectivity Pathfinder

The connectivity pathfinder creates a global connectivity map from the local connections layer. It assigns a connection value between 0 and 1 to each pair of points for each colour.

Local connections are assumed to be independent. Connection values of points outside the local connection window are computed as the product of the local connection values. Only the path resulting in the highest connection value is considered. The current implementation of the pathfinder runs Dijkstra’s shortest-path algorithm with each point as a starting point.

2.6 Connectivity-based Weight Selection

The simple eyes layer and second hidden layer are connected to the evaluation layer using connectivity-based weight selection.

Every neuron in the evaluation layer is connected to all neurons in the previous layer with weights depending on the connection value between the corresponding points predicted by the global connectivity layer. For that purpose connection values are transformed from the continuous values between 0 and 1 into 8 equally sized intervals\(^2\). For each colour \(c\) and pair of points \(p\) and \(p'\) the function \(\text{connection}(c, p, p') \in \{1...8\}\) returns the index of the interval.

Consider a neuron corresponding to a point \(p\) in the evaluation layer \(E\). The neuron has a bias weight \(b_{\text{type}}^E(p)\). Let \(L\) be one of the previous layers to which the evaluation layer is connected by connectivity-based weight selection (the simple eyes layer or second hidden layer), with \(n\) neurons per point. Then the

\(^2\)For efficiency, points with a connection value smaller than 0.1 were ignored.
neuron is connected to all neurons in the previous layer corresponding to points $p'$ and index $i \in \{1..n\}$ using the weights

$$w_{i,\text{type}(p),\text{type}(p'),\text{connection}(c,p,p')}^{EL}$$

for both colours $c$.

3. **Learning**

The learning is described according to the usual distinction between training (subsection 3.1) and testing (subsection 3.2).

3.1 **Training**

Games for training are produced by self-play. A move is selected by using 1-ply look-ahead with the sum of all outputs in the evaluation layer as the scoring function.

Although training on larger board sizes provides more reinforcement signal for each position, the 1-ply look-ahead would slow it down considerably. Therefore the experiments were done on a $9 \times 9$ board. However, the network architecture allows retraining the network on increasing board sizes to adapt it to the different ratios between edge and centre points.

For better exploration of the state space, in $15\%$ of the moves, instead of playing the move with the highest score, Gibbs sampling (Geman and Geman, 1984) over the move scores was used. The (unnormalised) probability of selecting a move with score $s$ was

$$P(s) = \exp(s/T)$$

with a temperature $T$ of 4.0. These positions were not trained.

After each played game, the 10 most recent games were trained using temporal-difference learning with $\lambda = 0$ (Sutton, 1988). The games were trained in random order going backward from the end position with immediate update of the weights after each position. The reason for the small value of $\lambda$ is that most parts of the network see only a portion of the board, so that the effective length of the game is not the number of moves in the global game, but is the number of moves in a part of the board.

The weights were updated by backpropagation. All neurons in layers that receive a reinforcement signal by the temporal difference algorithm were treated as output neurons in the backpropagation algorithm. The algorithmically computed connections to and from the global connectivity layer did not take part in the backpropagation algorithm.
3.2 Testing

After the first 100 games and every 5,000 games thereafter, the performance was tested by playing 100 games on a 9 \times 9 board against the program GNUGo version 3.0.0, released in 2001 (GNUGo, 2001). On the NNGS Go Server, the rating of GNUGo in 2001 was about 13 kyu (NNGS, 2001).

To obtain a variety of different games, every move of the network was selected by Gibbs sampling over the score with a temperature of 0.33. GNUGo always played White, the komi was 5.5. Identical games or games that could be mapped to other games by rotation and mirroring were sorted out.

The error of the mean value of the average score and percentage of wins is given by the standard deviation of the values divided by the square root of the number of games. However, this does not take into account partial correlations between the games. To get a more robust estimation of the error it is helpful to look at the deviation of the values late in the training process. At this time the changes in the weights of network are small, so that no big change in the playing strength is expected. From the reproducibility of the values between slightly different networks the error of the average score is estimated to be \pm 5 points and the error of the percentage of wins \pm10\%.

4. Experiments

The description of the experiments consists of two parts; the setup (subsection 4.1) and the results (subsection 4.2).

4.1 Setup

The size of the network was chosen to be 8 neurons per point in the first hidden layer and 2 neurons per point in the second hidden layer. The learning rate for the weight update was 3 \cdot 10^{-4}. The performance of the network was compared using two kinds of input.

**Raw board:** Only 1 neuron per point was used in the input layer with constant activation 1. This corresponds to providing the network only with the raw Go position as input, because the location of the stones is already used implicitly in the selection of the weights from the weight sets according to the point types.

**Preprocessed board:** The position was preprocessed and some local features of the position were used as input for the network. Only simple features that can be computed quickly and non-expensive tactical searches were used. The features included: number of stones and liberties of blocks, a weighted sum of higher-grade liberties (PON-estimation for blocks without any concept of groups (Tajima and Sanechika, 1998)) and the results of simple tactical searches (ladders (Sensei, 2003)). Also, basic
link patterns (straight 2 and 3 point jump, knight jump, long knight jump) were detected. See Table 1 for a detailed listing of the inputs.

<table>
<thead>
<tr>
<th>Input for empty points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...5</td>
</tr>
<tr>
<td>6...11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input for occupied points</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10...13</td>
</tr>
<tr>
<td>14...18</td>
</tr>
</tbody>
</table>

Table 1. Preprocessed input.

### 4.2 Results

The training took several weeks of CPU time on an Athlon XP 1800. Figure 4 shows the results of the test games against GNU GO. The network using the raw board input achieves an average score of about −25 points after 40,000 games. The network using the preprocessed input achieves an average score of about −5 points after 10,000 games.

Figures 5 and 6 show an example position with the evaluation output and the connectivity map for a point of the network using the preprocessed input. The network considers the left white group to be safe (0.2 is equivalent to 80% probability to become alive) but the centre group at F4 is unsafe (40% probability to become alive). The reason can be seen in the connectivity map for F4 in Figure 6: The probability for White to connect F4 to B4 is only 40%. 
Figure 4. Average score and wins against GNUGo. The error of the average score is estimated to be ±5 points and the error of the percentage of wins ±10% (see subsection 3.2).
Evaluation in Go by a Neural Network Using Soft Segmentation

A complete game of the network versus GNUGo is shown in Figure 7. GNUGo played Black in this game. The game was played with the network using preprocessed input after the training was finished. The network does a good job in keeping the black stones separated (with one mistake at move White 36) and wins by 8.5 points.
5. Conclusion

It was shown that the presented neural-network architecture can be successfully used for evaluating Go positions. Considering that the best Go programs currently play at a level around 8 kyu, the good performance against a 13 kyu program is promising. In particular, the approach addresses a weakness that current Go programs have in handling complicated tactical situations with many nearby weak groups. However, it is clear that a static evaluation cannot handle all kinds of positions. Thus, it will be necessary to add more local tactical search results to the input, and/or use the network as an evaluation function in a global search.

The most current version of NEUROGo uses the described architecture with more neurons in the hidden layers and more sophisticated input features. This increases the average score against GNUGo 3.0.0 to about +2 points and the percentage of wins to about 50%.

References

WHEN ONE EYE IS SUFFICIENT: A STATIC CLASSIFICATION

R. Vilà
Facultat de Matemàtiques i Estadística, UPC, Barcelona.
In stage in Labo IA, Université Paris 8.
ritx@ai.univ-paris8.fr

T. Cazenave
Labo IA, Université Paris 8, 2 rue de la Liberté, 93526, St-Denis, France.
cazenave@ai.univ-paris8.fr

Abstract A new classification for eye shapes is proposed. It allows to decide statically the status of the eye in some restricted conditions. The life property enables to decide when one eye shape is alive regardless the number of opponent stones inside. The method is easy to program and can replace a possibly deep search tree with a fast, reliable and static evaluation.

Keywords: computer Go, eye, neighbour classification, life property

1. Introduction

It is well known for both, Go players and Go programmers, that when a string has two eyes it is alive. Though sufficient, it is not a necessary condition. Sometimes one big eye is sufficient to live, either it is possible to make two eyes at any moment, or it is alive in seki.

This paper deals with the classification of large eyes and when one big eye is sufficient to live. Here we propose an algorithm that gives statically an answer to that question. It is easy to program and very fast. We present the neighbour classification, a completely new concept that enables to group eye shapes with common interesting properties. We also introduce the concept of life property that permits to decide when one eye shape is alive regardless the number of opponent stones inside. This property relies only on the shape of the eye and, when applicable, is very powerful. It is a completely safe tool as no heuristics are involved. It can be applied to a wide variety of situations.
Section 2 describes the existing work related to handling eyes and life and death. Section 3 sets accurate definitions of the concepts used throughout this paper. New concepts like end point or life property are proposed. We also have enlarged Müller’s (1999) concept of plain eye to cover statically more cases. Section 4 describes the main contribution of this paper, the neighbour classification and the theorem of the neighbour classification. Section 5 shows how to use the neighbour classification to identify the vital and end points for centre eyes. Section 6 discusses the limitations of the theorem for side and corner eyes and proposes possible ways to overcome them. Section 7 reports the application of this new theory to someai problems. Finally, we suggest that the reader has a quick look at the first two paragraphs of Section 4 before reading Section 3 so that the captions in the figures of this section are clarifying instead of confusing.

2. Previous Work

Several approaches have been made to life and death and eye characterisation with great success.

Landman (1996) applies combinatorial game theory to determine a value for a given eye space. Fotland (2002) describes the way his program, THE MANY FACES OF Go, analyzes eyes. He represents eye shapes as its game tree with four different values; the upper and lower bounds on the number of eyes, and two intermediate values aiming to include the effects of ko and uncertainty. This work deals mainly with a big variety of general eyes. He combines static analysis with a small search.


Big eyes are of great importance in a wide variety of someai problems. Though not being the key to the most common life-and-death problems, when they appear it is fundamental to handle them in a proper way. Most of the existing techniques treat them in an unsatisfactory way; either they treat them heuristically so unexpected situations may appear driving to a wrong answer, or they just let the search algorithm continue until they become a small eye with the subsequent inefficiency problems. Here we propose a theory and an algorithm to deal statically with this problem. It is very fast, easy to program, free of heuristic considerations and therefore completely reliable. It can replace completely a possibly deep search tree in a wide number of situations and it can be of great interest to enhance the existing techniques and to reduce the degree of inaccuracy.
3. Definitions

Eye. In this paper an eye\(^1\) will be an area completely surrounded by one block. Opponent and own stones will be allowed in the eye region and also empty points not adjacent to the surrounding block. This is a generalization of Müller's (1999) definition of plain eyes. We will classify eyes according to its position on the board.

**Corner Eye** — The eye contains a corner point and its two neighbours (cf. Figure 1).

![Figure 1. A corner [1122] eye, not plain.](image)

**Side Eye** — The eye is not a corner eye and contains at least three side points (note: a corner point is a particular case of side point) (cf. Figure 2).

![Figure 2. A side [112234] eye, plain.](image)

**Centre Eye** — All the eyes that are not corner or side eyes (note: the most part of big eye shapes can only be centre eyes (Mathworld, 2003)).

![Figure 3. A centre [1122233] eye, plain (left); a centre [112224] eye, plain (middle), and a centre [112224] eye, not plain (right).](image)

**Eye Shape.** This is the set of intersections of the eye. The intersections can be empty, or occupied by opponent or friendly stones. We will use the term Nakade Shape to refer to a set of intersections that, in case of being an Eye Shape, would have one or zero vital points.

**Eye Status.** We will define four possible status for a centre eye: Nakade, Unsettled, Alive, and AliveInAtari.

---

\(^1\)In the existing literature eye is used to refer to a small one-point eye, while bigger eyes are referred to as X-enclosed region (Benson, 1976) or Big eye (Fotland, 2002). In this paper we mainly deal with big eyes, therefore as no confusion is possible we will keep the term eye as we define it.
Nakade — the eye will end up as only one eye and this will not be sufficient to live. A nakade eye can be the result of: (1) an eye with an empty set of vital points (cf. Figure 4b) or (2) an eye with all the set of vital points filled by the opponent's stones (cf. Figure 4a).

![Figure 4a](image1) A nakade status for a [112233]-α eye. The two vital points are filled by the opponent.

![Figure 4b](image2) A nakade status for a [2222] eye. It has an empty set of vital points.

Unsettled — the eye can end up as a nakade eye or an alive eye depending on the colour to play. An unsettled eye is the result of an eye with one and only one empty intersection in the set of vital points (cf. Figure 5). An unsettled status is what Landman (1996) defines as $1 \frac{1}{2}$ ε.

Alive — the string owning the eye is alive no matter who plays first and no matter what the surrounding conditions are. An alive eye can be the result of: (1) an eye with two or more empty intersections in the set of vital points (cf. Figure 6a) or (2) the eye is a $n$-shape that cannot be filled by the opponent with a $(n - 1)$-nakade shape (Figure 6b). We will make no distinction between being alive or being alive in seki like in Figure 6b, as in many cases being alive in seki may be almost as good as living with two eyes (Landman, 1996).

![Figure 5](image3) An unsettled status for a [1222234] eye. One of the two vital points is empty (1).

![Figure 6a](image4) Alive status for a [1112234]-β eye. Even though these shape can be filled with a rabitty six, A and B belong to the set of vital points so we have a miai of life.

![Figure 6b](image5) Alive status for a [11222] eye. No matter how many stones plays White inside, Black is unconditionally alive. The opponent cannot fill the eye space with a nakade shape of size four.
**AliveInAtari** — this is a particular case in which the surrounding conditions determine the status of the eye. We say that an eye has an AliveInAtari status if there are only one or zero empty intersections adjacent to the surrounding block but capturing the opponent stones inside the eye grants an alive status. Only when the external liberties of the string owning the eye are played it is necessary to capture the stones inside the eye (cf. Figure 7a and 7b).

*Figure 7a.* AliveInAtari status for a [111223] eye. Capture grants life.

*Figure 7b.* AliveInAtari status for a [222233] eye. When A is played the status changes to Unsettled, and if both A and B are played the status is Nakade. However capturing the stones inside the eye grants an alive status.

**Vital Points.** A minimal set (one or more) of intersections inside the eye that should be filled by the opponent to grant a nakade status for the eye (cf. Figure 8a and 8b).

**End Points.** A minimal set (one or more) of intersections inside the eye that should not be filled by the opponent until the end to grant a nakade status for the eye in the process of killing the string (cf. Figure 8a and 8b).

This should not be confused with Fotland's (2002) number of ends. While Fotland's concept deals with the shape, our concept deals with the order in which the intersections of the eye should be filled by the opponent. In Figure 8a there is one end point but three Fotland's ends. However, in most cases we see that an end point from this paper's point of view is also a Fotland's end.

**Life Property.** We will say that an eye shape has the life property if the only possible status for this shape are Alive or AliveInAtari. Thus when an eye shape has the life property we only need to check whether the stones inside the eye should be captured due to an AliveInAtari status.

For example, a 3-shape in a line can have a Nakade, Unsettled, or AliveInAtari status depending on the opponent stones played inside. This 3-shape does not have the life property since a Nakade and Unsettled status are possible. In contrast, the [11222] shape showed in Figure 6b can only have an Alive status.
or an AliveInAtari status (when four out of the five intersections are played by the opponent), this shape has the life property. Therefore, while all the shapes having the life property are alive, not all the shapes having an alive status have the life property.

The life property should be regarded as a property slightly below Benson’s (1976) unconditional life, because if we have an AliveInAtari status it might be necessary to play inside the eye, but with the great advantage that detecting it is just a matter of counting neighbours as it will be shown in Section 4.

4. Neighbour Classification

Let $E_i$ be the set of all possible eye shapes of size $i$. Note that for $i = 1..6$ there is an isomorphism between $E_i$ and $P_i$ being $P_i$ the set of free $i$–polyominoes (Mathworld, 2003). An $n$–polyomino (or “$n$–mino”) is defined as a collection of $n$ squares of equal size arranged with coincident sides. Free polyominoes can be picked up and flipped, so mirror image pieces are considered identical. For size seven we should discard the holed-polyomino to keep the isomorphism.

Let $e \in E_i$, we define the Neighbour Classification of $e$, $NC(e)$, as a number of $i$ digits sorted from low to high; every intersection in the eye space is associated to a digit that indicates the number of neighbours (adjacent intersections) to that intersection that belong to the eye space (cf. Figure 9).

![Figure 9](image-url) For the rabbity six $NC(e) = 112224$. 
Let ~ be the following equivalence relation: let $e_1, e_2 \in \mathcal{E}_i$ then

$$e_1 \sim e_2 \iff NC(e_1) = NC(e_2)$$

Thus ~ gives a partition of $\mathcal{E}_i$ defined by the equivalence classes in $\mathcal{E}_i / \sim$ (cf. Appendix B).

**Example.** Given $\mathcal{E}_5$ we can find four different neighbour classifications for its elements (Note that $\|\mathcal{E}_5\| = \|\mathcal{P}_5\| = 12$) (Mathworld, 2003).

$$NC(e) \in \{112223, 111224, 111142, 122224\}, \quad \forall e \in \mathcal{E}_5$$

**THEOREM 1 (OF THE NEIGHBOUR CLASSIFICATION)** Let $e$ be a centre eye, $e \in \mathcal{E}_i$ and $[e] \in \mathcal{E}_i / \sim$ the equivalence class of $e$, for $i = 1..7$ if $e$ has the life property then $\forall f \in [e], f$ has the life property inversely if $e$ has not the life property then $\forall f \in [e], f$ has not the life property.

Proof: For $i \in \{1, 2, 3, 4\}$ there is no eye shape that has the life property so the theorem is correct. The 1–shapes and 2–shapes are always nakade, the two existing 3–shapes have one vital point so their status can be nakade or unsettled (depending on the fact whether the opponent has or has not played the vital point). There are five 4–shapes with zero, one or two vital points. All of them can have a nakade status if the opponent plays all the vital points. The interesting point comes with higher size shapes.

Under the conditions of the theorem, having the life property is just a matter of shape. If and only if an $i$–shape cannot be filled by the opponent with an $(i-1)$–shape that has one or zero vital points (Nakade Shape), then this $i$–shape has the life property.

Ko cannot arrive in the centre for eye shapes of size below seven. For size seven there are only two shapes that can have a ko status in the centre (cf. Figure 10). These shapes do not have the life property as they can be filled by a rabitty six so the ko does not interfere with our theorem.

---

**5–shapes.** There are two nakade shapes of size 4 (the square and the pyramid) so all the 5–shapes that do not contain a square or a pyramid will have the life
property. All the shapes belonging to \([11222]\) have the life property while the others do not. The only nakade shapes of size five are the bulky five and the star.

**6-shapes.** As before, all the 6-shapes that do not contain a bulky five or a star have the life property. These are the 26 shapes belonging to classes \([[112222], [111223], [111133]\}]. The only nakade shape of size six is the rabbity six.

**7-shapes.** Now we only have to care about those shapes containing a rabbity six, there are five shapes distributed in four equivalence classes. All the other 7-shapes have the life property. There is no nakade shape of size seven. This concludes the proof of Theorem 1. □

The exhaustive classification for all the eye shapes under size eight is summarized in Table 1.

<table>
<thead>
<tr>
<th>(\mathcal{E}_i)</th>
<th>(\mathcal{E}_i / \sim)</th>
<th>Life Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\mathcal{E}_1)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>([0])</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>(\mathcal{E}_2)</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>([11])</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>(\mathcal{E}_3)</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>([121])</td>
<td>2</td>
<td>No</td>
</tr>
<tr>
<td>(\mathcal{E}_4)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>([1122])</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>([1113])</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>([2222])</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>(\mathcal{E}_5)</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>([11222])</td>
<td>7</td>
<td>Yes</td>
</tr>
<tr>
<td>([11123])</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>([11114])</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>([12223])</td>
<td>1</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1. Neighbour Classification for \(\mathcal{E}_i\), \(i = 1 \ldots 7\).
The strength of the theorem lies in the fact that the *life property* only depends on the shape. So given an eye shape we have just to find its neighbour classification. If the class has the *life property*, whatever number of opponent stones inside, we know that the group owning the eye is alive, we only need to check if it is necessary to capture the stones inside the eye due to an AliveInAtari status. If the class does not have the *life property* a further study is required to decide the status (cf. Section 5).

We cannot extend the theorem for higher sizes in the centre because ko's and opponent eyes may appear. But we will see that usually it is not much of a problem as the *life property* is an excessively strong condition for such that eyes.

5. Vital Points and End Points Identification

Another interesting property of the *neighbour classification* is that it allows, for centre eyes, to find the *vital* and *end points* for a given eye shape just looking at its signature. Below we will show the identification for the five classes of size six without the *life property*. The identification for eye shapes with sizes from one to five is easy to find out and size seven requires an analogue procedure as size six.

For size six we have five different classes without the *life property* and thus, the status should be checked.

[[112224]] — The rabbity six is the only nakade shape of size six. The vital point is the 4-neighbour point. The 2-neighbour point not neighbouring the vital point may be considered an end point (cf. Figure 11). Though not necessary to be filled at the end, only if filled we should test for a non nakade shape inside.

[[111124]] — Vital points are \{2,4\}-neighbour points and the end point is the 1-neighbour point neighbouring the 2-neighbour point (cf. Figure 12).

[[222233]] — Vital points are the two 3-neighbour points (cf. Figure 13). There is no efficient way to define end points. So we should always test for a nakade four zigzag inside.

[[112233]] — We need to create two subclasses in this class to perform the identification. We define class [112233]-\(\alpha\) as the subset of two elements in class [112233] in which \{3,3\} are neighbours and the class...
[112233]-\(\beta\) as the subset in which \{3,3\} are not neighbours. So for \(\alpha\) elements are two vital points corresponding to the \{3,3\}-neighbour points and two end points corresponding to the \{1,1\}-neighbour points. For \(\beta\) elements the end points are the same, but all the other intersections are vital points (cf. Figure 14 and 15).

![Figure 14. Vital and end points for [112233]-\(\alpha\).](image)

![Figure 15. Vital and end points for [112233]-\(\beta\).](image)

[[122223]— The end point is the only 1-neighbour point. For vital points we need to consider the 3-neighbour point and its three neighbours (cf. Figure 16).

![Figure 16. Vital and end points for [122223].](image)

We do not know a unique way to find the vital and end points for no matter what kind of shape. So far a case by case implementation is needed, but in the process, the neighbour classification efficiently helps to determine them for each given class.

Once the identification is done it is possible to give the status and the hot point to play inside the eye, if necessary, depending on the opponent and friendly stones played in the eye shape (cf. Appendix A).

6. Corner and Side Eyes

To approach corner and side we should first remark the following implication (NoLP = No Life Property):

\[
\text{NoLP Centre} \Rightarrow \text{NoLP Side} \Rightarrow \text{NoLP Corner}
\]

Thus, once the study for centre eyes is done only classes with the life property in the centre need to be checked in the border and the corner.

For side eyes, theorem 1 continues to be true for sizes from one to four. For sizes five and six, ko only appears in classes that do not have the life property in the centre so the theorem continues to be true. For size seven there are two classes ([1222333] and [1112333]) that have the life property in the centre but fail to have it in the side due to ko situations. Unfortunately class [1122233]
has seven out of eleven members that do not have the life property due to ko while the other four continues to have it in the side (cf. Figure 17a and 17b). So the theorem is no longer applicable for side eyes with size seven.

Even though the theorem fails for side eyes, there is still a lot of knowledge that can be used for an implementation to solve side eyes. We will only have to consider more special cases. Shapes that do not have the life property will need to be treated more carefully in the side. For example, we have seen that a \([112233]\)-\(\alpha\) shape has two vital points. Therefore, if no vital point was played by the opponent, in the centre we had an alive status. This is no longer true in the side as Figure 18 shows. What might be called an Unsettled-Ko status appears for side eyes.

In the corner the situation is worse. Bent four in the corner, ko’s and the possibility for the opponent to make easily an eye inside the big eye makes the corner a difficult battleground to apply the theorem.

It is the moment to remark now how strong the condition of having the life property is. Strange examples of eye shapes in the corner can be found. They do not have the life property (ko’s can arrive) but they are almost impossible to kill in a real game (cf. Figure 19).

However, the fact that the theorem is not applicable in the corner does not mean that the neighbour classification is useless in those cases. It can be used to classify shapes in a straightforward way. For example, for size six there are only 12 out of 35 shapes that can be corner eyes. Six of these 12 are shapes of class \([112222]\) and \([111223]\). These shapes can have their status easily decided depending on the opponent stones played inside. For the other six shapes we can just return an unknown status and let the search continue until they become
a size five corner shape which are not so hard to decide by means of a case by case implementation.

7. Application to Semeai Problems

The *neighbour classification* has been successfully used and tested in semeai problems. Following Müller's (1999) classification of semeais and using the *neighbour classification* we have been able to solve statically classes from 0 to 2, but also all the semeais with centre and side eyes which are over class 2, either because the eye is not plain or because there are more than one non essential block inside the eye. This signifies an improvement over the results achieved statically and reported by Müller (1999).

A representative subset of semeai problems solved using the *neighbour classification* can be found at www.ai.univ-paris8.fr/rmx/semeai.zip.

8. Conclusions

Three new ideas about eyes are presented in this paper: the concept of *end point*, the definition of *life property*, and the *neighbour classification*.

The *neighbour classification* and the *life property* perform a completely safe tool for deciding eye status statically under some restricted conditions. The method is easy to program and can, in many situations, replace a possibly deep search tree with a fast, reliable and static evaluation.

For eye shapes that do not have the initial conditions, like side and corner eyes, we have shown that still a great deal of useful knowledge coming from the *neighbour classification* can be used.

It has been tested for semeai problems and proved to be a powerful tool.

References

Appendix A: Implementation for 6-shapes centre eyes

Below we present the general guidelines for an implementation of an algorithm to decide the status for size six centre eyes. We suppressed irrelevant details. Eye and Rzone should be regarded as classes that allow to store a set of intersections on the board. The names of the variables have been chosen to allow reading the implementation as if it were pseudo-code.

FindShapeVitalEnd takes e as input, decides the shape using the *neighbour classification* and initializes vital and end variables using the explanations already given in Section 5.

InitLocals takes e, vital and end as input and initializes EyeFilledSpace, vitalFilled and endFilled. The "Filled" variables contain the intersections in the eye, the vital zone and the end zone that are filled with opponent stones.

typedef enum eye6_t { t112224, t111124, t222233, t122223, t112233a, t112233b, other6 };

void Size6_Centre( Eye &e ) {
    eye6_t shape = other6;
    Rzone vital, vitalFilled, end, endFilled, EyeFilledSpace;

    FindShapeVitalEnd( &shape, &vital, &end, e);
    InitLocals( &EyeFilledSpace, &vitalFilled, &endFilled, vital, end, e);

    switch( shape ){
    case other6: //the shape has the life property
        if( EyeFilledSpace.size() == 5 )
            e.setEyeStatus( AliveInAtari );
        else
            e.setEyeStatus( Alive );
        break;
    case t111124:
        if( endFilled.size() == 0 ){
            switch( vitalFilled.size() ){
                case 0:
                    e.setEyeStatus( Alive );
                    break;
                case 1:
                    e.setEyeStatus( Unsettled );
                    //set Hot Spot: the intersection in vital not present in
                    //    vitalFilled
                    break;
                case 2:
                    e.setEyeStatus( Nakade );
                    break;
            }
        }
    }
else{
    if( EyeFilledSpace.size() == 5 )
        e.setEyeStatus( AliveInAtari );
    else
        e.setEyeStatus( Alive );
}
break;
case t122223:
    if( endFilled.size() == 1 ){
        if( EyeFilledSpace.size() == 5 )
            e.setEyeStatus( AliveInAtari );
        else
            e.setEyeStatus( Alive );
    }
else{
    switch ( vitalFilled.size() ){
    case 0:
    case 1:
        e.setEyeStatus( Alive );
        break;
    case 2:
        e.setEyeStatus( Alive );
        break;
    case 3:
        e.setEyeStatus( Unsettled );
        //set Hot Spot
        break;
    case 4:
        e.setEyeStatus( Nakade );
        break;
    }
    break;
}
case t222233: [...] 
    break;
case t112224: [...] 
    break;
case t112233a: [...] 
    break;
case t112233b: [...] 
    break;
}
Appendix B: Equivalence classes for \{5,6,7\}-shapes

Below we present the complete set of eye shapes of size five, six and seven grouped by equivalence classes.

*Figure 20.* The complete set of pentominoes grouped by classes.

*Figure 21.* The complete set of hexominoes grouped by classes.
Figure 22. The complete set of size seven eye shapes grouped by classes.
DF-PN IN GO: AN APPLICATION TO THE ONE-EYE PROBLEM

A. Kishimoto, M. Müller
Department of Computing Science University of Alberta
Edmonton, Canada, T6G 2E8
{kishi,mmueller}@cs.ualberta.ca, http://www.cs.ualberta.ca/~kishi/

Abstract
Search algorithms based on the notion of proof and disproof numbers have been shown to be effective in many games. In this paper, we modify the depth-first proof-number search algorithm df-pn, in order to apply it to the game of Go. We develop a solver for one-eye problems, a special case of enclosed tsume-Go [life and death] problems. Our results show that this approach is very promising.

Keywords: Go, proof-number search, df-pn, one-eye problem

1. Introduction

Computer Go is one of the ultimate challenges for games researchers. Despite a lot of efforts, the best programs can still be easily beaten even by human players of moderate skill.

One weakness of current Go programs is recognizing whether groups are alive or dead. Such tsume-Go (life and death) problems play a critical role in deciding the outcome of many games. Currently most Go-playing programs rely on a combination of exact and heuristic rules to evaluate tsume-Go (Chen and Chen, 1999; Kraszek, 1988). However, this approach does not always guarantee the correctness of the results.

In general, search is the only way to assess correctly the life-and-death status of stones. However, the large branching factor of Go makes it hard to apply a purely search-based approach. For enclosed tsume-Go problems with a small to moderate branching factor, the state of the art is already very good. GoTOOLS, the currently best tsume-Go solver, achieves high dan amateur level (Wolf, 2000). Search-based approaches have been very successful in other games such as chess, Othello, and shogi. In particular, in tsume-shogi (shogi checkmating problems), algorithms using proof and disproof numbers such as Seo’s PN* and Nagai’s df-pn (Seo, 1995; Nagai, 2002) have solved all difficult problems,
including those with solution sequences of hundreds of plies. Their performance far surpasses that of human players.

In this paper, we adapt the df-pn algorithm to the game of Go, and apply it to a restricted version of tsume-Go: the problem of making one eye in an enclosed position. This special case can be solved with a simpler evaluation function, but retains all the search-related difficulties of tsume-Go. To our knowledge, this is the first attempt to apply df-pn to computer Go. Our results are very promising. Even with very modest game-specific enhancements, our df-pn-based solver can quickly solve enclosed positions up to about 18 empty points. This compares favourably to state of the art tsume-Go solvers, which can solve general tsume-Go problems of up to about 14 empty points in reasonable time.

The structure of this paper is as follows. Section 2 describes the one-eye problem in Go and related work on tsume-Go. Section 3 reviews the df-pn algorithm. Section 4 explains a problem of df-pn in domains with position repetition, and develops a solution. Section 5 describes the basic one-eye solver and a few problem-specific enhancements. Section 6 deals with our current implementation of ko threats. Section 7 discusses the experimental results. Section 8 concludes and outlines further research directions.

2. The One-Eye Problem in Tsume-Go

The one-eye problem in Go is the question whether a player can create an eye connected to the player’s stones in a given region. Although the problem is simpler than full tsume-Go, it has many issues in common. For example, every tsume-Go problem in which the group under attack already has one eye in some region reduces to the one-eye problem on the rest of the board.

A specialized one-eye solver also promises to be useful to enhance the knowledge in a heuristic Go program. Typical current programs use elaborate heuristic rules to assign statically a number of eyes to a region of the board (Chen and Chen, 1999; Fotland, 2002). Replacing some of these heuristics by exact results can improve group strength estimation and thereby overall position evaluation.

A one-eye problem in a given Go position is defined by the following inputs.

- The two players, called the defender and the attacker. The defender tries to make an eye and the attacker tries to prevent it.
- The region, a subset of the board. At each turn, a player must either make a legal move within the region or pass.
- One or more blocks of crucial stones of the defender. The defender wins a one-eye problem by creating an eye connected to all the crucial stones inside the region. The attacker can win by either capturing at least one crucial stone, or by preventing the defender from creating an eye in the region.
- Safe attacker stones which surround the region together with crucial defender stones.
Figure 1 shows an example of a one-eye problem. Black is the defender and White is the attacker. Crucial stones are marked by triangles and the region is marked by crosses. Black must make an eye inside the region, while White tries to prevent that. There are unsafe stones at C6, E7, and H6. If these stones are captured, a player might play at such a point later, so they are part of the region.

2.1 Related Work on Tsume-Go

Wolf's (1994) GoToOLS is the currently best tsume-Go solver that specializes in solving completely enclosed positions. GoToOLS contains a sophisticated evaluation function that includes dynamic aspects, powerful rules for life-and-death recognition, and learning dynamic move ordering from the search (Wolf, 2000). Most competitive Go programs also contain a tsume-Go module. The commercial database Tsume-Go GOLIATH uses a proof-number search engine to check the user's inputs.

3. Df-pn: Depth-First Proof-Number Search

In this section we give an overview of the standard df-pn algorithm. Nagai's (2002) thesis is available for a detailed explanation.

3.1 Proof and Disproof Numbers

Proof and disproof numbers and Allis' proof-number search (PNS) (Allis, Van der Meulen, and Van den Herik, 1994) are the basis of this algorithm. The proof number of a node in an AND/OR tree is defined as the minimum number of leaf nodes that must be proven to prove the node for the first player, while the disproof number is the minimal number of leaf nodes that must be disproven (proven a win for the second player) in order to disprove the node. Proof and disproof numbers can be viewed as an estimate of how easy it is to prove or disprove a tree.
Proof-number search (PNS) maintains a proof number and a disproof number for each node. The leaf node to expand next is chosen in a best-first manner. Starting from the root, PNS traverses the tree by continuously selecting a child whose (dis)proof number is minimum at OR (AND) nodes, until it reaches a leaf node called a most-proving node. PNS expands that node and recomputes the proof and disproof numbers on the path to the root. This process continues until the root is either proven or disproven.

### 3.2 The Df-pn Algorithm

Df-pn (Nagai, 2002) turns PNS into a depth-first search algorithm by generalizing ideas behind Seo’s (1995) PN* algorithm. As a depth-first search, df-pn can expand less interior nodes and use a smaller amount of memory than PNS. Like PNS, it always expands a most-proving node.

Figure 2, adapted from Nagai (2002) presents pseudocode of the df-pn algorithm. Df-pn utilizes two thresholds, one for proof numbers and one for disproof numbers. For the sake of simplicity, the code is written in the negamax form, because disproof numbers are a dual notion of proof numbers. For each node \( n \), two variables \( \phi \) and \( \delta \) are defined as follows:

\[
\begin{align*}
 n.\phi &= \left\{ \begin{array}{ll}
 pn(n) & (n \text{ is an OR node}) \\
 dn(n) & (n \text{ is an AND node})
\end{array} \right. \\
 n.\delta &= \left\{ \begin{array}{ll}
 dn(n) & (n \text{ is an OR node}) \\
 pn(n) & (n \text{ is an AND node})
\end{array} \right.
\end{align*}
\]

While the iterative deepening method usually has a global threshold, df-pn’s thresholds work as local thresholds at each recursive call. This approach is similar to recursive best-first search (Korf, 1993). The main function Df-pn initializes both thresholds to infinity, and then calls the recursive function MID that iterates over nodes. When returning from MID, the root node is either proven or disproven. MID traverses the subtree below node \( n \) in a depth-first manner. It explores nodes while proof or disproof numbers do not exceed the threshold, or until it finds a terminal node that determines a winner. In the code, IsTerminal checks if \( n \) is a terminal node, while WinforCurrentNode checks whether a terminal node is a win or a loss. When a node \( n \) is expanded, the best child \( n_c \) in terms of proof and disproof numbers is selected by SelectChild for a recursive call to MID with the following new thresholds: \( n_c.\delta \) is set to the minimum of the current threshold for \( n \) and the value when \( n \)’s child with the second smallest \( \delta \) becomes the most-proving node during the exploration of \( n_c \)’s subtree. Note that \( n.\phi \) corresponds to \( n_c.\delta \) because of the negamax formulation. \( n_c.\phi \) works like the cost function of the IDA* algorithm (Korf, 1985).

Because df-pn is an iterative-deepening method that expands interior nodes again and again, the heart of the algorithm is the transposition table, a large
// Set up for the root node
int Df-pn(node r) {
    r.φ = oo;  r.δ = oo;
    MID(r);
    if (r.δ = oo)
        return win_for_root;
    else
        return loss_for_root;
}

// Select the most promising child
node SelectChild(node n, int &φc, int &δc, int &δ2) {
    node n_best;
    δc = φc = oo;
    for (each child n_child) {
        TTLookup(n_child.φ, δ);
        // Store the smallest and second
        // smallest δ in δc and δ2
        if (δ < δc) {
            n_best = n_child;
            δ2 = δc;  φc = φ;  δc = δ;
        }
        else if (δ < δ2)
            δ2 = δ;
        if (φ = oo)
            return n_best;
    }
    return n_best;
}

// Compute the smallest δ of
// n’s children
int ΔMin(node n) {
    int min = oo;
    for (each child n_child) {
        TTLookup(n_child.φ, δ);
        min = min(min, δ);
    }
    return min;
}

// Compute sum of φ of n’s children
int ΦSum(node n) {
    int sum = 0;
    for (each child n_child) {
        TTLookup(n_child.φ, δ);
        sum = sum + φ;
    }
    return sum;
}

// Select the most promising child
// Setting up for the root node
int Df-pn(node r) {
    r.φ = oo;  r.δ = oo;
    MID(r);
    if (r.δ = oo)
        return win_for_root;
    else
        return loss_for_root;
}

// Iterative deepening at each node
void MID(node n) {
    TTLookup(n.φ, δ);
    if (n.φ ≤ φ || n.δ ≤ δ) {
        // Exceed thresholds
        n.φ = φ;  n.δ = δ;
        return;
    }
    // Terminal node
    if (IsTerminal(n)) {
        if (WinForCurrentNode(n)) {
            n.φ = 0;  n.δ = oo;
            return;
        }
        else {
            n.φ = oo;  n.δ = 0;
            return;
        }
    }
    GenerateMoves(n);
    // Store larger proof and disproof
    // numbers to detect repetitions
    TTStore(n, n.φ, n.δ);
    // Iterative deepening
    while (n.φ > ΔMin(n) & &
        n.δ > ΦSum(n)) {
        n_c = SelectChild(n, φ_c, δ_c, δ2);
        // Update thresholds
        n_c.φ = n.δ + φ_c - ΦSum(n);
        n_c.δ = min(n.φ, δ2 + 1);
        MID(n_c);
    }
    // Store search results
    n.φ = ΔMin(n);  n.δ = ΦSum(n);
    TTStore(n, n.φ, n.δ);
}

Figure 2. Pseudocode of the df-pn algorithm.
cache storing previous search efforts, i.e., proof and disproof numbers for visited nodes. $TTstore$ stores proof and disproof numbers of a node in the table. $TTlookup$ checks the table for information on proof and disproof numbers of a node. If no result is found, both numbers are initialized to 1.

4. Computing Proof and Disproof Numbers in Domains with Repetitions

When we tried to apply df-pn to the one-eye problem in Go, df-pn could not solve some easy problems. The standard df-pn algorithm has a fundamental problem when applied to a domain with repetitions. Figure 3 shows an example. Assume $F$ is unknown, then the df-pn algorithm computes $\text{pn}(E) = \text{pn}(A) + \text{pn}(F)$. Hence, $\text{pn}(E)$ is larger than $\text{pn}(A)$. Df-pn’s termination condition is (see Figure 2):

$$n.\phi \leq \Delta \text{Min}(n) \lor n.\delta \leq \Phi \text{Sum}(n)$$

Usually the threshold of the proof number is only a little bit larger than $\text{pn}(A)$ when exploring $A$’s subtree in df-pn. Therefore, assuming that df-pn reaches $E$, df-pn exceeds the proof number threshold, stops expanding and updates $A$’s proof number to $\text{pn}(E) = \text{pn}(A) + \text{pn}(F)$. Even if $E$ is chosen in a later iteration, this phenomenon continues and $F$ is never explored. These repetitions often happen in Go, because passes are allowed. Two consecutive passes lead back the same position in a short loop.

Adding proof numbers from an ancestor to a node seems intuitively bad, since it leads to double-counting of the leaf nodes below. In our solution to this problem, we classify the children of a node into two types. A field minimal distance ($md$) of a node $n$ is initially set to the length of the shortest path from the root to $n$, the depth of $n$ in the search tree. We call a child $n_i$ normal if $n_i.md > n.md$, and old if $n_i.md \leq n.md$. Among the children $n_1 \cdots, n_k$ of $n$, let $n_1 \cdots, n_l$ ($1 \leq l \leq k$) be the normal and $n_{l+1} \cdots, n_k$ the old children. We modify the computation of proof and disproof numbers in the following way:

$$n.\phi = \min_{1 \leq i \leq k} n_i.\delta$$
\[ n.\delta = \begin{cases} 
\sum_{i=1}^{l} n_i \cdot \phi & \text{(if } \sum_{i=1}^{l} n_i \cdot \phi \neq 0) \\
\max_{l+1 \leq i \leq k} n_i \cdot \phi & \text{(if } \sum_{i=1}^{l} n_i \cdot \phi = 0) 
\end{cases} \]

Figure 4 illustrates an example of computing proof numbers. If \( F \) is neither proven nor disproven, then \( F \)'s proof number cannot be 0. Therefore we ignore \( A \) to compute \( E \)'s proof number, since \( A \) is an old child.

When a node has only old children, since all normal (and possibly some old) children have been solved, that node itself must be considered old, since now there is no way to prove or disprove it without exploring old nodes. Therefore, the \( md \) of that node must be updated. We set it to the minimum of the \( md \) fields of the currently unsolved old children.

Figures 5 and 6 depict an example of updating \( md \). In this figure, assuming that \( G \) is proven, \( E \) now has only an old child to explore, because \( F \) is also proven. In that case \( E \)'s minimal distance is updated to \( A \)'s distance, and \( pn(E) \) becomes \( pn(A) \). Further, \( C.md \) is set to \( E.md \) (see Figure 6). As a result, \( pn(C) \) is now ignored in the computation of \( pn(B) \), since \( C \) has become an old child.

Dealing with overcounting proof numbers caused by repetitions was essential to make df-pn work in Go. We note that Nagai (2002) achieves impressive

---

**Figure 4.** Df-pn with minimal distance \( md \).

**Figure 5.** Updating \( E \)'s minimal distance.

**Figure 6.** Computing \( C \)'s minimal distance.
results with his tsume-shogi solver, and described the GHI problem, which returns incorrect results involving cycles. However, this problem was not described in his papers. One possibility is that although the same problem could happen in shogi, it might happen much less often than in Go. Search in Go can easily return to identical states, for example by consecutive pass moves. Another possibility is that this problem tends to happen less frequently with additional search enhancements. Because Nagai’s tsume-shogi solver is enhanced with a great deal of domain-dependent knowledge, it might not occur in his case in practice. However, in a personal communication the existence of this problem in shogi was confirmed by Tsuruoka and Maruyama of team GEKISASHI. As well, Sakuta found that df-pn did not work better than PDS (Nagai, 1999) in his tsume-shogi solver, and gave as possible explanation the occurrence of DCGs (Sakuta, 2001).

5. **Application of Df-pn to the One-eye Problem**

Below we apply the df-pn algorithm to the one-eye problem. We start with the basic one-eye algorithm (5.1). Then we provide several game-specific search enhancements (5.2). The section is concluded by a simulation (5.3).

5.1 **The Basic One-eye Algorithm**

The basic algorithm, due to Anders Kierulf, is quite simple, and has been used as part of the tsume-Go search in the program EXPLORER for many years. It detects single-point eyes and false eyes.

The algorithm checks for all points in the region whether they are a potential eye point for the defender. Eyes are created by either surrounding empty points or by capturing attacker stones. If a safe eye connected to the crucial stones can be created in the region, the defender wins. If there is no potential eye space in the region, the attacker wins.

Whether a point \( E \) is a potential eye point is computed as follows:

- \( E \) occupied by unsafe attacker stone: *yes*.
- \( E \) occupied by safe attacker stone: *no*.
- \( E \) occupied by defender stone: *no*.
- \( E \) is empty: check the neighbours and the diagonal neighbours of \( E \).
  - Some direct neighbour is occupied by the attacker: *no*.
  - \( E \) is at the edge of the board and at least one diagonal neighbour contains a safe attacker: *no*.
  - At least two diagonal neighbours contain a safe attacker: *no*.
  - Otherwise: *yes*.

A potential eye point is a *safe eye* if all direct neighbours and all but one diagonal neighbour are occupied by defender stones. All diagonal neighbours are needed at the edge of the board. A safe eye is a defender win if the surrounding
block is connected to crucial stones, and all crucial stones are connected. The search generates all moves in the region, unless there are forced moves (see below).

5.2 Game-specific Search Enhancements

Safety by Connections to Safe Stones. Connectivity is a fundamental aspect of the game of Go. Most Go programs recognize connected blocks. We use connections to promote unsafe attacker stones to safe, and to prove that a defender eye is connected to crucial stones. Both types of connections help to reduce the search depth.

Our current implementation recognizes simple *mii* strategies (Müller, 1997) and some protected liberties for connections. Figure 7 gives examples of the strategy. In the left diagram, White has two ways (A and B) to connect. Even if Black plays first, the white block marked with squares can connect to safe stones. The stone at F6 is also safe now, because it has a connection either at C or at D. Since there is no eye space, this position can be statically evaluated as a loss for Black. Similarly, in the right diagram in Figure 7, the connection at E or F guarantees a win for Black. The algorithm to compute these connections is straightforward. It checks if safe blocks $S$ have two liberties to connect to a block $b$. If this is the case, $b$ is included in $S$ and the two liberties are marked to not be used for other connections. The process continues until no further blocks can be added to $S$.

![Figure 7](image.png) Connections to safe stones.
We find more safe stones by recognizing some forms of protected liberties. Figure 8 shows an example. The stone marked with a square has only one connection point at B to a safe white block. However, this connection is safe since the stone has another liberty and the opponent cannot play at B.

**Forced Moves.** Forced moves are a safe form of pruning when one player threatens to win immediately. We defined two kinds of forced moves, *forced attacker moves*, and *forced defender moves*, which correspond to ip1 or gi1 threats in Abstract Proof Search (APS) (Cazenave, 2002).

![Figure 8. Connection to safe stones on protected liberty.](image)

![Figure 9. Forced Moves.](image)

The first type of forced move is on a point where the defender could complete an eye that is connected to the crucial stones. The left position in Figure 9 presents an example. Black can make an eye at A. White must play at A to stop an immediate win for Black.

The second type of forced move is defined as follows:

1. There is no empty eye space for the defender in the region.
2 There is exactly one unsafe attacker's block \( b \).

3 \( b \) has a single-move connection to safe stones. If the defender plays any other move, the attacker can connect \( b \) to safety, leaving the defender with no potential eye points.

For instance, in the right position of Figure 9 the move at \( \text{B} \) is forced. Forced moves give a large reduction of the search space by decreasing the branching factor.

5.3 Simulation

Simulation was invented by Kawano (1996) to solve effectively positions with useless interposing piece drops in tsume-shogi problems. Later, Tanase (2000) extensively applied this idea to his \( \alpha \beta \)-search engine to reduce the overhead of calling the tsume-shogi solver inside the normal search. Assume that \( P \) is a proven position and \( Q \) is a “similar” one we want to prove. Simulation borrows moves from \( P \)’s proof tree to try to find a quick proof of \( Q \). A dual notion called dual simulation can be used to disprove a position.

In our solver, we apply simulation and dual simulation as follows:

- At an AND node \( n \), if one of \( n \)’s children, \( n_c \), is proven at some point in the search, apply simulation to all unsolved children of \( n \).
- Similarly, at an OR node \( n \), apply dual simulation if one of \( n \)’s children is disproven.

This use of simulation is much more extensive than in tsume-shogi. See the experimental section for a discussion.

6. Ko and Ko Threats

Sometimes the outcome of a one-eye problem depends on ko. It is therefore important to model ko threats and ko recaptures in the search algorithm.

The approach taken in GoTOOLS can require several searches (Wolf, 1994). The parameter to each search is how many ko recaptures are allowed for a specified kowinner.

Our current implementation allows only two options: one is to disallow any immediate ko recaptures; the other is to always allow ko recaptures for the designated kowinner. We search in one or two phases. The first search of a position, phase 1, disallows immediate ko recaptures, but marks nodes where such moves exist. If the search result depends on marked nodes, in phase 2 a re-search is performed. The loser of the phase 1 search is the designated kowinner for phase 2.

Phase 2 reuses the contents of the transposition table from phase 1. The following implementation of the transposition table aims to reduce the amount of re-search:
1 The Zobrist (1970) hash function is modified to account for a stone captured in the previous ko capture, to differentiate identical positions with different histories.

2 Two flags, one for each colour, in each transposition table entry keep track of any possible ko captures in the subtree below that node. If there is a ko capture for a player, the flag for the other player is set to indicate that we will allow a ko recapture after that node in a re-search. When a node \( n \) is proven (similarly for disproven), flags are set as follows:

- If \( n \) is an OR node and \( n_c \) is \( n \)'s proven child, \( n \)'s flags are set the same as \( n_c \)'s flags.
- If \( n \) is an AND node and the flag of one of the children is set, \( n \)'s flag is set. Otherwise, \( n \)'s flag is cleared.

In the phase 2 re-search, many phase 1 (dis)proofs can be reused. For example, assume that a node is proven and the flag for the kowinner is not set. Then we can use the proof from the transposition table. Similarly, we can also reuse disproofs. Even for nodes that are not proven or disproven, the proof and disproof numbers from phase 1 are valuable information for directing the re-search.

Re-searches usually have a low overhead, since we keep the previous results in the transposition table and reuse the table entries in most cases. However, if the solution changes dramatically by ko compared to the solution from the first search, a higher overhead results.

7. Empirical Results

This section consists of: test data (7.1), setup of experiments (7.2), test runs (7.3), and further comments on the experiments (7.4).

7.1 Test Data

In contrast to full tsume-Go, for which many large collections of test problems are available, we could not find any specialized collection of one-eye problems. Our current set of 70 test positions was created mainly by the authors. The problems can be played for both colours going first, resulting in a total of 140 problems. All problems are of the following form: a black group already has one safe eye, and is completely surrounded at a distance by safe white stones. The area in between forms the region, and the fate of the black group depends on whether it can form a second eye in the region. Problems of this kind are also suitable for solution by a general tsume-Go solver, since making one eye is equivalent to solving the tsume-Go problem.

The test set is available at http://www.cs.ualberta.ca/~games/go/oneeye. The problems include a mix of easy and hard problems. Some prob-
lems are challenging only for one colour playing first, and are very easy if the other colour plays first. Some of the positions are hard to solve for current tsume-Go programs. For an example, see Figure 10.

7.2 Setup of Experiments

All experiments were performed on a Pentium III/700 Mhz with a 100 MB transposition table. The time limit was 5 minutes per problem.

The following abbreviations are used for the methods and enhancements described above.

- **Df-pn**: The basic df-pn algorithm.
- **MIN**: Minimal distance modification for computing proof and disproof numbers.
- **AC**: Connections to safe stones for attacker
- **DC**: Connections to crucial stones for defender
- **FAM**: Forced attacker's moves
- **FDM**: Forced defender's moves
- **SIM**: Simulation and dual simulation

7.3 Test Runs

Adding Enhancements. Table 1 shows the results on the test set, starting with basic df-pn and switching on enhancements one by one. The total execution time and number of nodes expanded were computed using the subset of 126 problems that are all solved by methods (2) - (7) in the table.
A. Kishimoto, M. Müller

Table 1. Performance for successively switching on enhancements.

<table>
<thead>
<tr>
<th>Enhancements used</th>
<th>Number of problems solved</th>
<th>Total time (sec) 126 Problems</th>
<th>Total nodes expanded 126 Problems</th>
<th>Nodes expanded per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1): Df-pn</td>
<td>20</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(2): (1) + MIN</td>
<td>126</td>
<td>806</td>
<td>11,933,976</td>
<td>14,806</td>
</tr>
<tr>
<td>(3): (2) + AC</td>
<td>132</td>
<td>424</td>
<td>5,431,557</td>
<td>12,810</td>
</tr>
<tr>
<td>(4): (3) + DC</td>
<td>132</td>
<td>444</td>
<td>5,377,408</td>
<td>12,116</td>
</tr>
<tr>
<td>(5): (4) + FDM</td>
<td>132</td>
<td>436</td>
<td>5,142,100</td>
<td>11,802</td>
</tr>
<tr>
<td>(6): (5) + FAM</td>
<td>133</td>
<td>113</td>
<td>1,354,506</td>
<td>11,970</td>
</tr>
<tr>
<td>(7): (6) + SIM</td>
<td>134</td>
<td>81</td>
<td>1,168,683</td>
<td>14,347</td>
</tr>
</tbody>
</table>

The table shows the importance of the MIN modification. The only problems solved by basic df-pn were very easy ones that needed at most 400 nodes.

Search speed decreases a little with more enhancements, but improves again with simulation. Simulation provides a fast way to generate moves, faster than our current normal move generator, which has some overhead such as checking connections.

Leave-One-Out Experiments. The results for switching off a single enhancement at a time are shown in Table 2.

Table 2. Performance for turning off single enhancements.

<table>
<thead>
<tr>
<th>Enhancement Turned Off</th>
<th>Number of Problems Solved</th>
<th>Total Time (s) (129 Problems)</th>
<th>Total Nodes Expanded (129 Problems)</th>
<th>Nodes Expanded per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN</td>
<td>74</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AC</td>
<td>129</td>
<td>393</td>
<td>7,096,603</td>
<td>18,058</td>
</tr>
<tr>
<td>DC</td>
<td>134</td>
<td>138</td>
<td>2,081,344</td>
<td>15,070</td>
</tr>
<tr>
<td>FDM</td>
<td>134</td>
<td>393</td>
<td>7,096,603</td>
<td>18,058</td>
</tr>
<tr>
<td>FAM</td>
<td>133</td>
<td>402</td>
<td>5,950,111</td>
<td>13,907</td>
</tr>
<tr>
<td>SIM</td>
<td>133</td>
<td>175</td>
<td>2,123,969</td>
<td>12,137</td>
</tr>
</tbody>
</table>

Performance of Simulation. Table 3 shows the performance data for simulation in phase 1 searches. Since the method is applied in a very basic way, 45.2 % success seems to be a good initial result, with plenty of room for further refinements.

Table 3. Performance data on simulation for all 134 solved problems. All enhancements on. Phase 1 searches only.

<table>
<thead>
<tr>
<th>Total Nodes</th>
<th>Nodes by SIM</th>
<th>SIM calls</th>
<th>Successful calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>6,265,984</td>
<td>1,116,386 (17.8 %)</td>
<td>262,628</td>
<td>118,706 (45.2 %)</td>
</tr>
</tbody>
</table>
Re-searches for Ko. Table 4 shows a summary of the overhead incurred by re-searches for ko. In phase 1, immediate ko recaptures are not allowed. Phase 2 are the researches with a designated kowinner. The results in this table are also with all enhancements.

<table>
<thead>
<tr>
<th>Total Nodes (134 Problems)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 1</td>
<td>Phase 2</td>
</tr>
<tr>
<td>6,265,984 (95.4 %)</td>
<td>304,107 (4.6 %)</td>
</tr>
</tbody>
</table>

Table 4. Overheads for ko re-searches

The overhead is quite small, but of course this is mainly a property of the test set used, which contains only a few cases with complex ko fights. In the worst case encountered, problem oneeyeb.10.sgf with Black to play, phase 1 took 7,340 nodes and phase 2 took 11,728 nodes.

7.4 Further Comments on the Experiments

Reexpansion of Interior Nodes. One concern in df-pn is the overhead of reexpansion of interior nodes. In our experiments, the ratio of interior nodes expanded to total nodes is about 30 %. In Seo’s experiments in shogi, this ratio was about 20 %. Since information achieved dynamically is usually more reliable than static evaluations, we think that our 30 % is still a very small price to pay to achieve more cut-offs.

Currently Unsolved Problems. Our solver currently cannot solve 6 problems in our test suite. Figure 11 shows an example. All unsolved problems feature large regions with many possible moves. Besides, some problems such as in Figure 10 and 11 stretch the limits of the one-eye problem, such as semeai, and tsume-Go. Figure 11, for example, can be seen as a problem whether white stones adjacent to black crucial stones can make two-eyes or not, having no split between the one-eye and tsume-Go problems. As well, the practical limit of our current solver seems to be at around 18 empty points, which compares favourably with about 14 empty reported for GoTools. However, we need further investigations to assess this limit and improve the ability of our solver for difficult problems.
8. Conclusions and Future Work

The early results of our work on applying df-pn to Go and specifically to the one-eye problem are very encouraging. There are numerous possible enhancements, both for improving the search algorithm and for adding Go-specific knowledge. Examples are recognizing larger eyes, refining the knowledge about connections, generalizing forced moves similar to Cazenave’s APS, heuristic initialization of proof and disproof numbers, and search in open-ended areas.

To apply these ideas to other problems in Go is also an interesting research topic. Examples include full tsume-Go (two-eye problems), tactical capture search and connection search.

8.1 Comparison with related Programs

We would like to compare our program with general tsume-Go solvers to assess its performance. However, it is hard to make a fair comparison since our algorithm solves only a restricted problem. Evaluation for two eyes is much harder than for one eye, and many years of hard work have gone into the development of the Go knowledge in programs such as GoTOOLS. However, we believe that as a search algorithm our modified df-pn works very well for Go. In informal experiments it seems that our algorithm can already solve harder problems in our test set than other programs. One possible advantage of the df-pn algorithm is that it uses the transposition table more extensively. Only solved positions are saved in the transposition table in GoTOOLS (Wolf, 2000), while in df-pn proof and disproof numbers of previous iterations are stored in the transposition table to improve the order of tree expansion (Nagai, 2002).

8.2 The GHI Problem in Df-pn

So far in this paper, we have not addressed the graph history interaction (GHI) problem (Palay, 1985). This problem occurred in our experiments, for example in double or triple ko situations. If GHI is ignored, incorrect results are stored in the transposition table. We developed a new approach that differs from the one described in Breuker et al. (2001) for the case of proof-number search. The method will be described in a forthcoming publication (Kishimoto and Müller, 2003).

Acknowledgments

Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).
References


LEARNING TO SCORE FINAL POSITIONS IN THE GAME OF GO

E.C.D. van der Werf, H.J. van den Herik, J.W.H.M. Uiterwijk
Institute for Knowledge and Agent Technology, Department of Computer Science,
Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands
{e.vanderwerf,herik,uiterwijk}@cs.unimaas.nl, http://www.cs.unimaas.nl/rvanderwerf/

Abstract This paper presents a learning system for scoring final positions in the Game of Go. Our system learns to predict life and death from labelled game records. 98.9% of the positions are scored correctly and nearly all incorrectly scored positions are recognized. By providing reliable score information our system opens the large source of Go knowledge implicitly available in human game records, thus paving the way for a successful application of machine learning in Go.

Keywords: Go, learning, neural net, scoring, game records, life and death

1. Introduction

Evaluating Go positions is one of the hardest tasks in Artificial Intelligence (AI). In the last decades, stimulated by Ing’s million-dollar price for the first computer program to defeat a professional Go player (which has expired unchallenged), Go has received significant attention from AI research (Bouzy and Cazenave, 2001; Müller, 2002). Yet, despite all efforts, the best computer Go programs are still no match even for human amateurs of only moderate skill. Partially this is due to the complexity of Go, which makes brute-force search techniques infeasible on the $19 \times 19$ board. However, on the $9 \times 9$ board, which has a complexity between Chess and Othello (Bouzy and Cazenave, 2001), the current Go programs perform nearly as bad. The main reason lies in the lack of good positional evaluation functions. Many (if not all) of the current top programs rely on (huge) static knowledge bases derived from the programmers’ Go skills and Go knowledge. As a consequence the top programs are extremely complex and difficult to improve. In principle a learning system should be able to overcome this problem.

In the past decade several researchers have used machine-learning techniques in Go. After Tesauro’s (1995) success story many researchers, including Dahl (2001), Enzenberger (1996) and Schraudolph et al. (1994), have applied Tem-
poral Difference (TD) learning for learning evaluation functions. Although TD-learning is a promising technique, which was underlined by NEUROGO's latest performance at the 21st Century Championship Cup (Myers, 2002), there has not been a major breakthrough, such as in Backgammon, and we believe that this will remain unlikely to happen in the near future as long as most learning is done from self-play or against weak opponents.

Over centuries humans have acquired extensive knowledge of Go. Since that knowledge is implicitly available in the games of human experts, it should be possible to apply machine-learning techniques to extract that knowledge from game records. So far game records have only been used successfully for move prediction (Enderton, 1991; Dahl, 2001; van der Werf et al., 2002). However, we are convinced that much more can be learned from these game records.

One of the best sources of game records on the Internet is the No Name Go Server game archive (NNGS, 2002). NNGS is a free on-line Go club where people from all over the world can meet and play Go. All games played on NNGS since 1995 are available on-line. Although NNGS game records contain a wealth of information, the automated extraction of knowledge from these games is a non-trivial task at least for the following three reasons.

**Missing Information.** Life-and-death status of blocks is not available. In scored games only a single numeric value representing the difference in points is available.

**Unfinished Games.** Not all games are scored. Human games often end by one side resigning or abandoning the game without finishing it, which often leaves the status of large parts of the board unclear.

**Bad Moves.** During the game mistakes are made which are hard to detect. Since mistakes break the chain of optimal moves it can be misleading (and incorrect from a game-theoretical point of view) to relate positions before the mistake to the final outcome of the game.

The first step toward making the knowledge in the game records accessible is to obtain reliable scores at the end of the game. Reliable scores are obtained by correct classification of life-and-death stones on the board. This paper focuses on determining life and death for final positions. By focusing on final positions we avoid the problem of unfinished games and bad moves during the game, which will have to be dealt with later.

It has been pointed out by Müller (1997) that proving the score of final positions is a hard task. For a set of typical human final positions, Müller showed that a combination of complex static analysis and search, still leaves around 75% of the board-points unproven. Heuristic classification of his program EXPLORER classified most blocks correctly, but still left some regions unsettled (and to be played out further). Although this may be appropriate for computer-computer games it can be annoying in human-computer games, especially under the Japanese rules which penalize playing more stones than necessary.
Since proving the score of most final positions is not (yet) an option, we focus on learning a heuristic classification. We believe that a learning algorithm for scoring final positions is important because: 1) it provides a more flexible framework than the traditional hand-coded static knowledge bases, and 2) it is a necessary first step toward learning to evaluate non-final positions. In general such an algorithm is good to have because: 1) large numbers of game records are hard to score manually, 2) publicly available programs still make too many mistakes scoring final positions, and 3) it can avoid unnecessarily long human-computer games.

The rest of this paper is organised as follows. Section 2 discusses the scoring method. Section 3 presents the learning task. Section 4 introduces the representation. Section 5 provides details about the dataset. Section 6 reports our experiments. Finally, section 7 presents our conclusions.

2. The Scoring Method

The two main scoring methods in Go are territory scoring and area scoring. Territory scoring, used by the Japanese rules, counts the surrounded territory plus the number of captured opponent stones. Area scoring, used by the Chinese rules, counts the surrounded territory plus the alive stones on the board. The result of the two methods is usually the same up to one point. The result may differ since one player placed more stones than the other, for three possible reasons; (1) because Black made the first and the last move, (2) because one side passed more often during the game, and (3) because of handicap stones. (Under Japanese rules the score may also differ because territory surrounded by alive stones in seki is not counted.) In this paper area scoring is used since it is the simplest scoring method to implement for computers.

Area scoring works as follows: First, the life-and-death status of blocks of connected stones is determined. Second, dead stones are removed from the board. Third, each empty point is marked Black, White, or neutral (the non-empty points are already marked by their colour). The empty points can be marked by flood filling or by distance. Flood filling recursively marks empty points to their adjacent colour. In the case that a flood fill for Black overlaps with a flood fill for White the overlapping region becomes neutral. (As a consequence all non-neutral empty regions must be completely enclosed by one colour.) Scoring by distance marks each point based on the distance toward the nearest remaining black or white stone(s). If the point is closer to a black stone it is marked black, if the point is closer to a white stone it is marked white, otherwise (if the distance is equal) the point does not affect the score and is marked neutral. Finally, the difference between black and white points, together with a possible komi, determines the outcome of the game.
In final positions scoring by flood filling and scoring by distance should give the same result. If the result is not the same, there are large open regions with unsettled interior points, which usually means that some stones should have been removed or some points could still be gained by playing further. Comparing flood filling with scoring by distance is therefore a useful check to detect whether the game is finished and scored correctly.

3. The Learning Task

The task of learning to score comes down to learning to determine which blocks of connected stones are dead and should be removed from the board. This is learned from a set of labelled final positions, for which the labels contain the colour controlling each point. A straightforward implementation would be to learn to classify all blocks based on the labelled points. However, for some blocks this not a good idea because their status can be irrelevant and forcing them to be classified just complicates the learning task.

The only blocks required for a correct score are either alive and at the border of their area, or dead in the opponent’s area. This is illustrated by Figure 1. Here all marked stones must be classified. The stones marked by triangles must be classified alive. The stones marked by squares must be classified dead. The unmarked stones are irrelevant for scoring because they are not at the border of their area and their possible capturability does not affect the score. For example, the two black stones in the top-left corner kill the white block and are in Black’s area. However, they can always be captured by White, so forcing them to be classified as alive or dead is misleading and even unnecessary. (The stones in the bottom left corner are alive in seki because neither side can capture. The two white stones in the upper right corner are adjacent to two neutral points and therefore also at the border of White’s region.)

3.1 Recursion

Usually blocks of stones are not alive on their own. Instead they form chains or groups which are only alive in combination with other blocks. Their status also may depend on the status of neighbouring blocks of the opponent, i.e., blocks can live by capturing the opponent. (Although one might be tempted to conclude that life and death should be dealt with at the level of groups this does not really help because the human notion of a group is not well defined, difficult to program, and may even require an underlying notion of life and death.)
Because life and death of blocks is strongly related to the life and death of other blocks the status of other (usually nearby) blocks has to be taken into account. Partially this can be done by including features for nearby blocks in the representation. In addition, it seems natural to consider a recursive framework for classification which employs the predictions for other blocks to improve performance iteratively. In our implementation this is done by training a cascade of classifiers which use previous predictions for other blocks as additional input features.

4. Representation

In this section we will present the representation of blocks for classification. Several representations are possible and used in the field. The most primitive representations typically employ the raw board directly. A straightforward implementation is to concatenate three bitboards into a feature vector, for which the first bitboard contains the block to be classified, the second bitboard contains other friendly blocks and the third bitboard contains the enemy blocks. Although this representation is complete, in the sense that all relevant information is preserved it is unlikely to be efficient because of the high dimensionality and lack of topological structure.

4.1 Features for Block Classification

A more efficient representation employs a set of features based on simple measurable geometric properties, some elementary Go knowledge and some hand-crafted specialised features. Several of these features are typically used in Go programs to evaluate positions (Chen and Chen, 1999; Fotland, 2002). The features are calculated for single friendly and opponent blocks, multiple blocks in chains, and colour-enclosed regions (CERs).

For each block our representation consists of the following features: (All features are single scalar values unless stated otherwise.)

- **Size** measured in occupied points.
- **Perimeter** measured in number of adjacent points, including points over the edge.
- **Opponents** are the occupied adjacent points.
- **(First order) liberties** are the free adjacent points.
- **Protected liberties** are the liberties which cannot be played by the opponent, because of suicide or being directly capturable.
- **Auto-atari liberties** are liberties which by playing them reduce the liberties of the block from 2 to 1, which means that the blocks would become directly capturable (such liberties are protected for an adjacent opponent block).
Second-order liberties are the liberties of (first-order) liberties (excluding the first-order liberties).

Third-order liberties are the liberties of second-order liberties (excluding first- and second-order liberties).

Adjacent opponent blocks

Local majority is the number of opponent stones minus the number of friendly stones within a Manhattan distance of 2 from the block.

Centre of mass represented by the distance to the closest and second-closest edge.

Bounding box size is the number of points in the smallest rectangular box that can contain the block.

Adjacent to each block are colour-enclosed regions. CERs consist of connected empty and occupied points, surrounded by stones of one colour or the edge. It is important to know whether an adjacent CER is fully accessible, because a fully accessible CER surrounded by safe blocks provides at least one sure liberty. To detect fully accessible regions we use so-called miai strategies as applied by Müller (1997). In contrast to Müller’s original implementation we also add miai accessible interior empty points to the set of accessible liberties, and also use protected liberties for the chaining. For fully accessible CERs we include:

- Number of regions
- Size
- Perimeter
- Split points are crucial points for preserving connectedness in the local 3 x 3 window around the point. (The region could still be connected by a big loop outside the local 3 x 3 window.)

For partially accessible CERs we include:

- Number of partially accessible regions
- Accessible size
- Accessible perimeter
- Size of the unaccessible interior.
- Perimeter of the unaccessible interior.
- Split points of the unaccessible interior.

The size, perimeter and number of split points are summed for all regions. We do not address individual regions because the representation must have a fixed number of features, whereas the number of regions is not fixed.

Another way to analyse CERs is to look for possible eyespace. Points forming the eyespace should be empty or contain capturable opponent stones. Empty points directly adjacent to opponent stones are not part of the eyespace. Points on the edge with one or more diagonally adjacent alive opponent stones and
points with two or more diagonally adjacent alive opponent stones are false eyes. False eyes are not part of the eyespace (we ignore the unlikely case where a big loop upgrades false eyes to true eyes). Initially we assume all diagonally adjacent opponent stones to be alive. However, in the recursive framework (see below) the eyespace is updated based on the status of the diagonally adjacent opponent stones after each iteration. For directly adjacent eyespace of the block we include:

- Size
- Perimeter

Since we are dealing with final positions it is often possible to use the optimistic assumption that all blocks with shared liberties can form a chain (during the game this assumption is dangerous because the chain may be split). For this, so-called, optimistic chain we include:

- Number of blocks
- Size
- Perimeter
- Split points
- Adjacent CERs
- Adjacent CERs with eyespace
- Adjacent CERs, fully accessible from at least one block.
- Size of adjacent eyespace
- Perimeter of adjacent eyespace
- External opponent liberties are liberties of adjacent opponent blocks which are not accessible from the optimistic chain.

Adjacent to the block in question there may be opponent blocks. For the weakest (measured by the number of liberties) directly adjacent opponent block we include:

- Perimeter
- Liberties
- Shared liberties
- Split points
- Perimeter of adjacent eyespace

The same features are also included for the second-weakest directly adjacent opponent block and the weakest opponent block directly adjacent to or sharing liberties with the optimistic chain of the block in question.

By comparing a flood fill starting from Black with a flood fill starting from White we find unsettled empty regions which are disputed territory (assuming all blocks are alive). If the block is adjacent to disputed territory we include:

- Direct liberties in disputed territory.
- Liberties of all friendly blocks in disputed territory.
- Liberties of all enemy blocks in disputed territory.
4.2 Additional Features for Recursive Classification

For the recursive classification the following six additional features are used:

- *Predicted value* of the strongest friendly block with a shared liberty.
- *Predicted value* of the weakest adjacent opponent block.
- *Predicted value* of the second-weakest adjacent opponent block.
- *Average predicted value* of the weakest opponent block's optimistic chain.
- *Adjacent eyespace size* of the weakest opponent block's optimistic chain.
- *Adjacent eyespace perimeter* of the weakest opponent block's optimistic chain.

Next to these additional features the predictions are also used to update the eyespace, i.e., dead blocks can become eyespace for the side that captures, alive blocks cannot provide eyespace, and diagonally adjacent dead opponent stones are not counted for detecting false eyes.

5. The Data Set

In the experiments we used game records obtained from the NNGS archive (NNGS, 2002). All games were played on the 9×9 board between 1995 and 2002. We only considered games which are played to the end and scored, thus ignoring unfinished or resigned games. Since the game records only contain a single numeric value for the score, we had to find a way to label all blocks.

5.1 Scoring the Data Set

For scoring the dataset we initially used a combination of GNUGo and manual labelling. Although GNUGo has the option to finish games and label blocks the program could not be used without human supervision. The reasons for this are bugs, the inherent complexity of the task, and the mistakes made by weak human players which ended the game in positions that are not final, or scored them incorrectly. Fortunately, nearly all mistakes are easily detected by comparing GNUGo's scores and labelled boards with the numeric scores stored in the game records. As an extra check all boards containing open regions with unsettled interior points (where flood filling does not give the same result as distance-based scoring) were also inspected manually.

Since the scores did not match in many positions the labelling proved to be very time consuming. We therefore only used GNUGo to label the games played in 2002 and 1995. With the 2002 games a classifier was trained. When we tested the performance on the 1995 games it outperformed GNUGo's labelling. So therefore our classifier replaced GNUGo for labelling all other games (1996-2001), retraining it each time a new year was labelled. Although this speeded up the process it still required a fair amount of human intervention mainly because of games that contained incorrect scores in their game record.
A few hundred games had to be thrown out completely because they were not finished, contained illegal moves, contained no moves at all (for at least one side), or both sides were played by the same player. In a small number of cases, where the last moves would have been trivial but not actually played, we made the last few moves manually.

Eventually we ended up with a dataset containing 18,222 final positions. Around 10% of these games were scored incorrectly (by the players) and were inspected manually. (Actually the number of games we inspected is significantly higher because of the games that were thrown out and because our initial classifiers and GNU Go made mistakes). On average the final positions contained 5.8 alive blocks, 1.9 dead blocks, and 2.7 irrelevant blocks. (In the case that one player gets the full board all his blocks were assumed irrelevant although at least one block should of course be classified as alive.)

Since the Go scores on the 9×9 board range from −81 to +81 the chances of an incorrect labelling leading to a correct score are low, nevertheless it could not be ruled out completely. On inspecting an additional 1% of the positions randomly we found none that were labelled incorrectly. Finally, when all games were labelled, we re-inspected all positions for which our best classifier seemed to predict an incorrect score. This final pass detected 42 positions (0.2%) which were labelled incorrectly, mostly because our initial classifiers had made the same mistakes as the players that scored the games.

5.2 Statistics

Since many game records contained incorrect scores we looked for reasons and gathered statistics. The first thing that came to mind is that weak players might not know how to score. Therefore in Figure 2 the percentage of incorrectly scored games related to the strength of the players is shown. (Although in each game only one side may have been responsible for the incorrect score, we always assigned blame to both sides.) The two marker types distinguish between rated and unrated players. Although unrated players have a value for their rating, it is an indication given by the player and not by the server. Only after playing sufficiently many games the server assigns players a rating.

Although a significant number of games are scored incorrectly this is usually not considered a problem when the winner is correct. (Players typically forget to remove some stones when they are far ahead.) Figure 3 shows how often incorrect scoring by rated players converts a win to a loss.

It should be noted that the percentages in Figures 2 and 3 were weighted over all games regardless of the player. Therefore they do not necessarily reflect the probabilities for individual players, i.e., the statistics can be dominated by a small group of players that played many games. This group at least contains some computer players which have a tendency to get robbed of their points.
in the scoring phase. We therefore also calculated some statistics that were normalised over individual players. For rated players the average probability of scoring a game incorrectly is 4.2%, the probability of cheating (the incorrect score converts loss to win) is 0.66%, and the probability of getting cheated is 0.55%. For unrated players the average probability of scoring a game incorrectly is 11.2%, the probability of cheating is 2.1%, and the probability of getting cheated is 1.1%. The fact that the probability of getting cheated is lower than the probability of cheating is the result of a small group of players (several of which are computer programs) that systematically lose points in the scoring phase, and a larger group of players that take advantage of them.

6. Experiments

In this section experimental results are presented for: (1) selecting a classifier, (2) performance of the representation, (3) recursive performance, (4) full board performance, and (5) performance on the 19\times19 board. Unless stated otherwise the various training and validation sets, used in the experiments, were extracted from games played between 1996 and 2002. The test set was always the same, containing 7149 labelled blocks extracted from 919 games played in 1995.

6.1 Selecting a Classifier

An important choice is selecting a good classifier. In pattern recognition there is a wide range of classifiers to choose from (Jain et al., 2000). We tested a number of well-known classifiers for their performance on datasets of 100, 1000, and 10000 examples. The classifiers are: Nearest Mean Classifier (NMC), Linear Discriminant Classifier (LDC), Logistic Linear Classifier (LOGLC), Quadratic Discriminant Classifier (QDC), Nearest Neighbour Classifier (NNC), K-Nearest Neighbours Classifier (KNNC), BackPropagation Neural net Classifier with momentum and adaptive learning (BPNC), Levenberg-Marquardt Neural net Classifier (LMNC), and RProp Neural net Classifier (RPNC). Some preliminary experiments with a Support Vector Classifier, Decision Tree Clas-
sifiers, a Parzen classifier and a Radial Basis Neural net Classifier were not pursued further because of excessive training times and/or poor performance. All classifiers except the neural net classifiers, for which we directly used the standard matlab toolbox, were used as implemented in PRTools3 (Duin, 2000).

The results, shown in Table 1, indicate that performance first of all depends on the size of the training set. The linear classifiers perform better than the quadratic classifier and nearest neighbour classifiers. For large datasets training KNNC is very slow because it takes a long time to find an optimal value of the parameter $k$. The number of classifications per second of (K)NNC is also low because of the large number of distances that must be computed (all training examples are stored). Although the performance of the nearest neighbour classifiers might be improved by editing and condensing the dataset, we did not investigate them further.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Training size</th>
<th>Training error (%)</th>
<th>Test error (%)</th>
<th>Training time (s)</th>
<th>Classi. speed ($s^{-1}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMC</td>
<td>100</td>
<td>2.8</td>
<td>3.9</td>
<td>0.0</td>
<td>$4.9 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>4.0</td>
<td>3.8</td>
<td>0.1</td>
<td>$5.2 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>3.8</td>
<td>3.6</td>
<td>0.5</td>
<td>$5.3 \times 10^4$</td>
</tr>
<tr>
<td>LDC</td>
<td>100</td>
<td>0.7</td>
<td>3.0</td>
<td>0.0</td>
<td>$5.1 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>2.1</td>
<td>2.0</td>
<td>0.1</td>
<td>$5.2 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>2.2</td>
<td>1.9</td>
<td>0.9</td>
<td>$5.3 \times 10^4$</td>
</tr>
<tr>
<td>LOGLC</td>
<td>100</td>
<td>0.0</td>
<td>9.3</td>
<td>0.2</td>
<td>$5.2 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.0</td>
<td>2.6</td>
<td>1.1</td>
<td>$5.2 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>1.0</td>
<td>1.2</td>
<td>5.6</td>
<td>$5.1 \times 10^4$</td>
</tr>
<tr>
<td>QDC</td>
<td>100</td>
<td>0.0</td>
<td>13.7</td>
<td>0.1</td>
<td>$3.1 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>1.0</td>
<td>2.1</td>
<td>0.1</td>
<td>$3.2 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>1.9</td>
<td>2.1</td>
<td>1.1</td>
<td>$3.2 \times 10^4$</td>
</tr>
<tr>
<td>NNC</td>
<td>100</td>
<td>0.0</td>
<td>18.8</td>
<td>0.0</td>
<td>$4.7 \times 10^3$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.0</td>
<td>13.5</td>
<td>4.1</td>
<td>$2.4 \times 10^2$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.0</td>
<td>10.2</td>
<td>$4.1 \times 10^3$</td>
<td>$2.4 \times 10^2$</td>
</tr>
<tr>
<td>KNNC</td>
<td>100</td>
<td>7.2</td>
<td>13.1</td>
<td>0.0</td>
<td>$4.8 \times 10^3$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>4.2</td>
<td>4.4</td>
<td>$1.0 \times 10^1$</td>
<td>$2.4 \times 10^2$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>2.8</td>
<td>2.8</td>
<td>$9.4 \times 10^3$</td>
<td>$2.6 \times 10^2$</td>
</tr>
<tr>
<td>BPNC</td>
<td>100</td>
<td>0.5</td>
<td>3.6</td>
<td>2.9</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.2</td>
<td>1.5</td>
<td>$1.9 \times 10^4$</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.5</td>
<td>1.0</td>
<td>$1.9 \times 10^2$</td>
<td>$1.9 \times 10^4$</td>
</tr>
<tr>
<td>LMNC</td>
<td>100</td>
<td>2.2</td>
<td>7.6</td>
<td>$2.6 \times 10^4$</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.7</td>
<td>2.8</td>
<td>$3.2 \times 10^3$</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.5</td>
<td>1.2</td>
<td>$2.4 \times 10^3$</td>
<td>$1.9 \times 10^4$</td>
</tr>
<tr>
<td>RPNC</td>
<td>100</td>
<td>1.5</td>
<td>4.1</td>
<td>1.4</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>0.2</td>
<td>1.7</td>
<td>7.1</td>
<td>$1.8 \times 10^4$</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
<td>0.4</td>
<td>1.1</td>
<td>$7.1 \times 10^1$</td>
<td>$1.9 \times 10^4$</td>
</tr>
</tbody>
</table>

Table 1. Performance of classifiers.
The best classifiers are the neural network classifiers. It should however be noted that their performance may be slightly overestimated with respect to the size of the training set, because we used an additional validation set to stop training (this was not possible for the other classifiers because they are not trained incrementally). The Logistic Linear Classifier performs nearly as good as the neural network classifiers, which is quite an achievement considering that it is just a linear classifier.

The results of Table 1 were obtained with neural networks that employed one hidden layer containing 15 neurons with hyperbolic tangent sigmoid transfer functions. Since our choice for 15 neurons was quite arbitrary a second experiment was performed in which we varied the number of neurons in the hidden layer. In Figure 4 results are shown for the RPNC. The classification errors marked with triangles represent results for training on 5,000 examples, the stars indicate results for training on 15,000 examples. The solid lines are measured on the independent test set, whereas the dash-dotted lines are obtained on the training set. The results show that even moderately sized networks easily overfit the data. Although the performance initially improves with the size of the network, it seems to level off for networks with over 50 hidden neurons (the standard deviation is around 0.1 %). Again clearly the key factor in improving performance is in increasing the training set.

![Figure 4. Sizing the neural network for the RPNC.](image)

### 6.2 Performance of the Representation

In section 4 we claimed that a raw board representation is inefficient for predicting life and death. To validate this claim we measured the performance of such a representation and compared it to our specialised representation.

The raw representation consists of three concatenated bitboards, for which the first bitboard contains the block to be classified, the second bitboard contains other friendly blocks and the third bitboard contains the enemy blocks. To remove symmetry the bitboards are rotated such that the centre of mass of the block to be classified is always in a single canonical region.

Since high-dimensional feature spaces tend to raise several problems which are not directly caused by the quality of the individual features we also tested two compressed representations. These compressed representations were generated by performing Principal Component Analysis (PCA) on the raw representation.
For the first PCA mapping the number of features was chosen identical to our specialised representation. For the second PCA mapping the number of features was set to preserve 90% of the total variance.

The results, shown in Table 2, are obtained for the RPNC with 15, 35, and 75 neurons in the hidden layer, for training sets with 100, 1,000 and 10,000 examples. All values are averages over 11 runs with different training sets, validation sets (same size as the training set), and random initialisations. The errors, measured on the test set, indicate that a raw representation alone requires too many training examples to be useful in practice. Even with 10,000 training examples the raw representation performs much weaker than our specialised representation with only 100 training examples. Simple feature-extraction methods such as Principal Component Analysis do not seem to improve performance, indicating that preserved variance of the raw representation is relatively insignificant for determining life and death.

<table>
<thead>
<tr>
<th>Training Size</th>
<th>Extractor</th>
<th>Test error 15 neurons (%)</th>
<th>Test error 35 neurons (%)</th>
<th>Test error 75 neurons (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>-</td>
<td>29.1</td>
<td>26.0</td>
<td>27.3</td>
</tr>
<tr>
<td>100</td>
<td>pca1</td>
<td>22.9</td>
<td>22.9</td>
<td>22.3</td>
</tr>
<tr>
<td>100</td>
<td>pca2</td>
<td>23.3</td>
<td>24.3</td>
<td>21.9</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>13.7</td>
<td>13.5</td>
<td>13.4</td>
</tr>
<tr>
<td>1000</td>
<td>pca1</td>
<td>16.7</td>
<td>16.2</td>
<td>15.6</td>
</tr>
<tr>
<td>1000</td>
<td>pca2</td>
<td>14.2</td>
<td>14.5</td>
<td>14.4</td>
</tr>
<tr>
<td>10000</td>
<td>-</td>
<td>7.5</td>
<td>6.8</td>
<td>6.5</td>
</tr>
<tr>
<td>10000</td>
<td>pca1</td>
<td>9.9</td>
<td>9.3</td>
<td>9.1</td>
</tr>
<tr>
<td>10000</td>
<td>pca2</td>
<td>8.9</td>
<td>8.2</td>
<td>7.7</td>
</tr>
</tbody>
</table>

Table 2. Performance of the raw representation.

### 6.3 Recursive Performance

Our recursive framework for classification is implemented as a cascade of classifiers which use extra features, based on previous predictions as discussed in subsection 4.2, as additional input. The performance measured on an independent test set for the first 4 steps is shown for various sizes of the training set in Table 3. The results are averages of 5 runs with randomly initialised networks containing 50 neurons in the hidden layer (the standard deviation is around 0.1%).

The results show that recursive predictions improve the performance. However, the only significant improvement comes from the first iteration. The improvements are by far not significant for the average 3- and 4-step errors. The reason for this is that sometimes the performance got stuck or even worsened after the first iteration. Preliminary experiments suggest that large networks
were more likely to get stuck after the first iteration than small networks, which might indicate some kind of overfitting. A possible solution to overcome this problem is to retrain the networks a number of times, and pick the best based on the performance on the validation set. If we do this the best networks, trained on 100,000 training examples, achieve a 4-step error of 0.25%.

<table>
<thead>
<tr>
<th>Training Size</th>
<th>Direct error (%)</th>
<th>2-step error (%)</th>
<th>3-step error (%)</th>
<th>4-step error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>1.93</td>
<td>1.60</td>
<td>1.52</td>
<td>1.48</td>
</tr>
<tr>
<td>10,000</td>
<td>1.09</td>
<td>0.76</td>
<td>0.74</td>
<td>0.72</td>
</tr>
<tr>
<td>100,000</td>
<td>0.68</td>
<td>0.43</td>
<td>0.38</td>
<td>0.37</td>
</tr>
</tbody>
</table>

*Table 3. Recursive performance.*

### 6.4 Full Board Performance

So far we have concentrated on the percentage of blocks that are classified correctly. Although this is an important measure it does not directly tell us how often boards will be scored correctly (a board may contain multiple incorrectly classified blocks). Further we do not yet know what the effect is on the score in number of board points. Therefore we tested our classifier on the full-board test positions (which were not used for training or validation).

For our best 4-step classifier trained with 100,000 examples we found that 1.1% of the boards were scored incorrectly. For 0.5% of the boards the winner was not identified correctly. The average number of incorrectly scored board points (using distance-based scoring) was 0.15, however in case a board is scored incorrectly this usually affects around 14 board points (which counts double in the numeric score).

### 6.5 Performance on the 19×19 Board

The experiments presented above were all performed on the 9×9 board which, as was pointed out before, is a most challenging environment. Nevertheless, it is interesting to test whether the techniques scale up to the 19×19 board. So far we did not have the time to label large quantities of 19×19 games. So, training directly on the 19×19 board was not an option. Despite of this we tested our classifiers, which were trained from blocks observed on the 9×9 board, on the problem set *IGS_31.counted* from the Computer Go Test Collection. This set contains 31 labelled 19×19 games played by amateur dan players, and was used by Müller (1997). On the 31 final positions our 4-step classifier classified 5 blocks incorrectly (0.5% of all relevant blocks), and as a consequence 2 final positions were scored incorrectly. The average number of incorrectly scored board points was 2.1 (0.6%).
In his paper Müller stated that heuristic classification of his program EXPLORER classified most blocks correctly. Although we do not know the exact performance of EXPLORER we believe it is safe to say that our system, which scored 99.4% of all board points correctly, is performing at least at a comparable level. Furthermore, since our system was not even trained explicitly for 19 × 19 games there may still be significant room for improvement.

7. Conclusions

We have developed a system that learns to score final positions from labelled examples. On unseen game records our system scored around 98.9% of the positions correctly without any human intervention. Compared to the average rated player on NNGS (who for scored 9 × 9 games has a rating of 7 kyu) our system is more accurate at removing all dead blocks, and performs comparable on determining the correct winner.

By comparing numeric scores and counting unsettled interior points we can efficiently detect nearly all incorrectly scored final positions. Although some final positions are assessed incorrectly by our classifier, most are in fact scored incorrectly by the players. Detecting these games is important because most machine-learning methods require reliable training data for good performance.

7.1 Future Work

By providing reliable score information our system opens the large source of Go knowledge which is implicitly available in human game records. The next step will be to apply machine learning in non-final positions. We believe that the representation and techniques presented in this paper provide a solid basis for static predictions in non-final positions.

The good performance of our system was obtained without any search, indicating that static evaluation is sufficient for most human final positions. Nevertheless, we believe that some (selective) search can still improve the performance. Adding selective features that involve search and integrating our system into MAGOG, our 9×9 Go program, will be an important next step.

Although the performance of our system is already quite good for labelling game records, there are, at least in theory, still positions which may be scored incorrectly when our classifier makes the same mistake as the human players. Future work should determine how often this happens in practice.

Another point where our system can be improved is the representation. Although the current representation performs adequately, some features may be redundant or correlated. Feature extraction, feature selection, and possibly adding some new features may improve performance even further.
References


NNGS (2002). The no name go server game archive.


MONTE-CARLO GO DEVELOPMENTS

B. Bouzy
Université Paris 5, UFR de mathematiques et d’informatique, C.R.I.P.5, 45, rue des Saints-Pères 75270 Paris Cedex 06 France
bouzy@math-info.univ-paris5.fr

B. Helmstetter
Université Paris 8, laboratoire d’Intelligence Artificielle
2, rue de la Liberte 93526 Saint-Denis Cedex France
bh@ai.univ-paris8.fr

Abstract We describe two Go programs, OLGA and OLEG, developed by a Monte-Carlo approach that is simpler than Bruegmann’s (1993) approach. Our method is based on Abramson (1990). We performed experiments to assess ideas on (1) progressive pruning, (2) all moves as first heuristic, (3) temperature, (4) simulated annealing, and (5) depth-two tree search within the Monte-Carlo framework. Progressive pruning and the all moves as first heuristic are good speed-up enhancements that do not deteriorate the level of the program too much. Then, using a constant temperature is an adequate and simple heuristic that is about as good as simulated annealing. The depth-two heuristic gives deceptive results at the moment. The results of our Monte-Carlo programs against knowledge-based programs on 9x9 boards are promising. Finally, the ever-increasing power of computers lead us to think that Monte-Carlo approaches are worth considering for computer Go in the future.

Keywords: Monte-Carlo approach, computer Go, heuristics

1. Introduction

We start with two observations. First, when termination of the search process of a game tree is possible, the process provides the best move and constitutes a proof on that best move. The process does not necessarily need domain-dependent knowledge but its cost is exponential in the search depth. Second, a domain-dependent move generator generally yields a good move, but without any verification. It costs nothing in execution time but the move generator remains incomplete and always contains errors. When considering the game of Go, these two observations are crucial. Global tree search is not possible in Go and knowledge-based Go programs are very difficult to improve (Bouzy and Cazenave, 2001). Therefore, this paper explores an intermediate approach in
which a Go program performs a global search (not a global tree search) using very little knowledge. This approach is based on statistics or more specifically, on Monte-Carlo methods. We believe that such an approach does neither have the drawback of global tree search with very little domain-dependent knowledge (no termination), nor the drawback of domain-dependent move generation (no verification). The statistical global search described in this paper terminates and provides the move with a kind of verification. In this context, the paper claims the adequacy of statistical methods, or Monte-Carlo methods, to the game of Go.

To support our view, Section 2 describes related work about Monte-Carlo methods applied to Go. Section 3 focuses on the main ideas underlying our work. Then, Section 4 highlights the experiments to validate these ideas. Before conclusion, Section 5 discusses the relative merits of the statistical approach and its variants along with promising perspectives.

2. Related Work

At a practical level, the general meaning of Monte Carlo lies in the use of the random generator function, and for the theoretical level we refer to Fishman (1996). Monte-Carlo methods have already been used in computer games. In incomplete information games, such as poker (Billings et al., 2002), scrabble (Sheppard, 2002), and backgammon (Tesauro, 2002), this approach is natural: because the information possessed by your opponent is hidden, you want to simulate this information. In complete information games, the idea of replacing complete information by randomized information is less natural. Nevertheless, it is not the first time that Monte-Carlo methods have been tried in complete information games. This section deals with two previous contributions (Abramson, 1990; Bruegmann, 1993).

2.1 Abramson’s Expected-Outcome

Evaluating a position of a two-person complete information game with statistics was tried by Abramson (1990). He proposed the expected-outcome model, in which the proper evaluation of a game-tree node is the expected value of the game’s outcome given random play from that node on. The author showed that the expected outcome is a powerful heuristic. He concluded that the expected-outcome model of two-player games is “precise, accurate, easily estimable, efficiently calculable, and domain-independent”. In 1990, he tried the expected-outcome model on the game of 6x6 Othello. The ever-increasing computer power enables us to use this model now for Go programs.

2.2 Bruegmann’s Monte-Carlo Go

Bruegmann (1993) was the first to develop a Go program based on random games. The architecture of the program, Gobble, was remarkably simple. In
order to choose a move in a given position, Gobble played a large number of almost random games from this position to the end, and scored them. Then, he evaluated a move by computing the average of the scores of the random games in which it had been played.

This idea is the basis of our work. Below we describe some issues of Gobble. In our work, described in Section 3, they are subject to improvements.

1. **No filling of the eyes.** Moves that filled one’s eyes were forbidden. This was the sole domain-dependent knowledge used in Gobble. In the game of Go, the groups must have at least two eyes in order to be alive (with the relatively rare exception of groups living in seki). If the eyes could be filled, the groups would never live and the random games would not actually finish. However, the exact definition of an eye has its importance.

2. **Evaluation of the moves.** Moves were evaluated according to the average score of the games in which they were played, not only at the beginning but at any stage of the game, provided that it was the first time one player had played at the intersection. This was justified by the fact that moves are often good independently of the stage at which they are played. However, this can turn out to be a fairly dangerous assumption.

3. **Selection of the moves.** Moves were not chosen completely randomly, but rather on their current evaluation, good moves having more chances to be played first. Moreover, simulated annealing was used to control the probability that a move could be played out of order. The amount of randomness put in the games was controlled by the temperature; it was set high at the beginning and gradually decreased. Thus, in the beginning, the games were almost completely random, and at the end they were almost completely determined by the evaluations of the moves. However, we will see that both are possible: (1) to fix the temperature to a constant value, and (2) to make the temperature even infinite, which means that all moves are played with equal probability.

### 3. Our Work

This section first describes the basic idea underlying our work (Subsection 3.1). Then, it presents our Go programs, Olga and Oleg (Subsection 3.2). The only important domain-dependent consideration of the method, the definition of eyes, is described in Subsection 3.3. Finally, in Subsection 3.4 a graph explaining the various possible enhancements to the basic idea is given.

#### 3.1 Basic Idea

Though the architecture of the Gobble program was particularly simple, some points were subject to discussion. Our own algorithm for Monte-Carlo Go programs is an adaptation of Abramson’s (1990). The basic idea is: to evaluate
a position by playing a given number of completely random games to the end - without filling the eyes - and then scoring them. The evaluation corresponds to the mean of the scores of those random games. Choosing a move in a position means playing each of the moves and maximize the evaluations of the positions obtained at depth 1.

3.2 Two Programs: OLGA and OLEG

We developed two Go programs based on the basic idea above: OLGA and OLEG. OLGA and OLEG are far-fetched French acronyms for “ALEatoire GO” or “aLEatoire GO” that mean random Go. OLGA was developed by Bouzy (2002) as a continuation of the INDIGO development. The main idea was to use an approach with very little domain-dependent knowledge. At the beginning, the second idea in the OLGA development was to concentrate on the speed of the updating of the objects relative to the rules of the game, which was not highlighted in the previous developments of INDIGO. Of course, OLGA uses code available in INDIGO.

OLEG was written by Helmstetter. Here, the main idea was to reproduce the Monte-Carlo Go experiments of Bruegmann (1993) to obtain a Go program with very little Go knowledge. OLEG uses the basic data structure of GNUGo that is already very well optimized by the GNUGo team (Bump, 2003).

Both in OLEG and in OLGA, the quality of play depends on the precision expected that varies with the number of tests performed. The time to carry out these tests is proportional to the time spent to play one random game. On a 2 GHz computer, OLGA plays 7,000 random 9x9 games per second and OLEG 10,000.

Because strings, liberties, and intersection accessibilities are updated incrementally during the random games, the number of moves per second is almost constant and the time to play a game is proportional to the board size. Since the precision of the expected value depends on the square of the number of random games, there is no need to gain 20 per cent in speed, which would only bring about a 10-per-cent improvement in the precision. However, optimizing the program very roughly is important. A first pass of optimizations can gain a ratio of 10, and the precision can be three times better in such a case, which is worthwhile.

OLGA and OLEG share the basic idea and most of the enhancements that are described in Subsection 3.4. They are used to test the relative merits of each enhancement. However, each program uses its own definition of eyes.

3.3 How to Define Eyes?

The only domain-dependent knowledge required is the definition of an eye. It is important for the random program not to play a move in an eye. Without
this rule, the random player would never make living groups and the games would never end. There are different ways to define “eyes” as precisely as possible with domain-dependent knowledge such as Fotland (2002) and Chen and Chen (1999). Our definitions are designed to be integrated into a random Go-playing program; they are simple and fast but not correct in some cases.

In OLGA, an eye is an empty intersection surrounded by stones of one colour with two liberties or more.

In OLEG, an eye is an empty intersection surrounded by stones belonging to the same string.

The upside of both definitions is the speed of the programs. OLEG’s definition is simpler and faster than OLGA’s. Both approaches have the downside of being wrong in some cases. OLEG’s definition is very restrictive: OLEG’s eyes are actual true eyes but it may fill an actual eye surrounded by more than one string. Besides, OLGA has a fuzzy and optimistic definition: it never fills an actual eye but, to connect its stones surrounding an OLGA’s eye, OLGA always expects one adjacent stone to be put into atari.

3.4 Various Possible Enhancements

So far, we have identified a few possible enhancements from the basic idea. They are shown in Figure 1. This figure also shows the enhancements used by OLEG and OLGA in their standard configurations. Two of the enhancements were already present in Gobble, namely the all moves as first heuristic (which means making statistics not only for the first move but for all moves of the random games) and simulated annealing. For the latter, an intermediate possibility can be adopted: instead of making the temperature vary during the game, we make it constant.

With a view of speeding up the basic idea, an alternative to the all-moves-as-first heuristic is progressive pruning of which only the first move of the random games is taken into account for the statistics, and the moves of which the evaluation is too low compared to the best move are pruned.

Making a minimax at depth 2 and evaluating the positions by making random games from this position naturally evolves from the basic idea. The expected result is an improvement of the program reading ability. For instance, it would suppress moves that work well only when the opponent does not respond.

4. Experiments

Starting from the basic idea, this section describes and evaluates the various enhancements: progressive pruning, all-moves-as-first heuristic, temperature, simulated annealing, and depth-two enhancements.

For each enhancement, we set up experiments to assess its effect on the level of our programs. One experiment consists in a match of 100 games between the
program to be assessed and the experiment reference program, each program playing 50 games with Black. In most experiments, the program to be assessed is a program in which one parameter varies, and the reference program is the same program with the parameter fixed to a reference value. In the other set of experiments, the program to be assessed uses the enhancement while the reference program does not. The result of an experiment is generally a set of relative scores provided by a table assuming that the program of the column is the max player. Given that the standard deviation of 9x9 games played by our programs is roughly 15 points, 100 games enable our experiments to lower $\sigma$ down to 1.5 points and to obtain a 95% confidence interval of which the radius equals $2\sigma$, i.e., 3 points. We have used 2 GHz computers. When the response time of the assessed program varies with the experimental parameters, we mention it. Furthermore, all programs in this work do not use any conservative or aggressive style depending on who is ahead in a game, they only try to maximize their own score. The score of a game is more significant than the winning percentage which is consequently not included in the experiments’ results. We terminate this section with an assessment of OLGA and OLEG against two existing knowledge-based programs INDIGO and GNUGO, in showing the results of an all-against-all tournament.

### 4.1 Progressive Pruning

As contained in the basic idea, each move has a mean value $m$, a standard deviation $\sigma$, a left expected outcome $m_l$ and a right expected outcome $m_r$. For a move, $m_l = m - \sigma r_d$ and $m_r = m + \sigma r_d$. $r_d$ is a ratio fixed up by practical experiments. A move $M_1$ is said to be statistically inferior to another move $M_2$ if $M_1.m_r < M_2.m_l$. Two moves $M_1$ and $M_2$ are statistically equal when $M_1.\sigma < \sigma_e$ and $M_2.\sigma < \sigma_e$ and no move is statistically inferior to the other. $\sigma_e$ is called standard deviation for equality, and its value is determined by experiments.
In Progressive Pruning (PP), after a minimal number of random games (100 per move), a move is pruned as soon as it is statistically inferior to another move. Therefore, the number of candidate moves decreases while the process is running. The process stops either when there is only one move left (this move is selected), or when the moves left are statistically equal, or when a maximal threshold of iterations is reached. In these two cases, the move with the highest expected outcome is chosen. The maximal threshold is fixed to 10,000 multiplied by the number of legal moves. This progressive pruning algorithm is similar to the one described in Billings et al. (2002).

Due to the increasing precision of mean evaluations while the process is running, the max value of the root is decreasing. Consequently, a move can be statistically inferior to the best one at a given time and not later. Thus, the pruning process can be either hard (a pruned move cannot be a candidate later on) or soft (a move pruned at a given time can be a candidate later on). Of course, soft PP is more precise than hard PP. Nevertheless, in the experiments shown here, OLGA uses hard PP.

<table>
<thead>
<tr>
<th>$r_d$</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0</td>
<td>+5.6</td>
<td>+7.3</td>
<td>+9.0</td>
</tr>
<tr>
<td>time</td>
<td>10'</td>
<td>35'</td>
<td>90'</td>
<td>150'</td>
</tr>
</tbody>
</table>

Table 1. Times and relative scores of PP with different values of $r_d$, against PP($r_d=1$).

<table>
<thead>
<tr>
<th>$\sigma_e$</th>
<th>0.2</th>
<th>0.5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>0</td>
<td>-0.7</td>
<td>-6.7</td>
</tr>
<tr>
<td>time</td>
<td>10'</td>
<td>9'</td>
<td>7'</td>
</tr>
</tbody>
</table>

Table 2. Times and relative scores of PP with different values of $\sigma_e$, against PP($\sigma_e=0.2$).

The inferiority of one move compared to another, used for pruning, depends on the value of $r_d$. Theoretically, the greater $r_d$ is, the less pruned the moves are, and, as a consequence, the better the algorithm performs, but the slower it plays. The equality of moves, used to stop the algorithm, is conditioned by $\sigma_e$. Theoretically, the smaller $\sigma_e$ is, the fewer equalities there are, and the better the algorithm plays but with an increased slowness. We set up experiments with different versions of OLGA to obtain the best compromise between the time and the level of the program. The first set of experiments consisted in assessing the level and speed of OLGA depending on $r_d$. OLGA($r_d$) played a set of games either with black or white against OLGA($r_d=1$). Table 1 shows the mean of the relative score of OLGA($r_d$) when $r_d$ varies from 1 up to 8. Both the minimal number of random games and the maximal threshold remain constant (100 and 10,000 respectively).

This experiment shows that $r_d$ plays an important role in the move pruning process. Large values of $r_d$ correspond to the basic idea. To sum up, progressive pruning loses little strength compared to the basic idea, between five or ten points according to the value of $r_d$. In the next experiments, $r_d$ is set to 1. The second set of experiments deals with $\sigma_e$ in the same way. Table 2 shows the mean of the relative score of OLGA($\sigma_e$) when $\sigma_e$ varies from 0.2 up to 1.
OLGA($\sigma_e=1$) yields the worst score while using less time. This experiment confirms the role played by $\sigma_e$ in the move pruning process. In the next experiments, $\sigma_e$ is set to 0.2.

4.2 The All-Moves-As-First Heuristic

When evaluating the terminal position of a given random game, this terminal position may be the terminal position of many other random games in which the first move and another friendly move of the random game are reversed. Therefore, when playing and scoring a random game, we may use the result either for the first move of the game only, or for all moves played in the game as if they were the first to be played. The former is the basic idea, the latter is what was performed in GOBBLE, and we use the term all moves as first heuristic.

4.2.1 Advantages and Drawbacks. The idea is attractive, because one random game helps evaluate almost all possible moves at the root. However, it does have some drawbacks because the evaluation of a move from a random game in which it was played at a late stage is less reliable than when it is played at an early stage. This phenomenon happens when captures have already occurred at the time when the move is played. In figure 2 the values of the moves $A$ for Black and $B$ for White largely depend on the order in which $\text{fuey}$ are played.

There might be more efficient ways to analyse a random game and decide whether the value of a move is the same as if it was played at the root. Thus, we would obtain the best of both worlds: efficiency and reliability. To this end, at least one easy thing should be done (it has already been done in GOBBLE and in OLEG): in a random game, if several moves are played at the same place because of captures, modify the statistics only for the player who played first.

The method has another troublesome side-effect: it does not evaluate the value of an intersection for the player to move but rather the difference between the values of the intersection when it is played by each player. Indeed, in most random games, any intersection will be played either by one player or the other, with an equal probability of about $1/2$ (an intersection is almost always played at least once during a random game). Therefore, the average score of all random games lies approximately in the middle between the average score when White has played a move and the average score when Black has played a move. Most often, this problem is not serious, because the value of a move for one player
is often the same for both players; but sometimes it is the opposite. In Figure 3 the point C is good for White and bad for Black. On the contrary D and E are good for Black only.

4.2.2 Experimental Comparison with Progressive Pruning. Compared to the very slow basic idea the gain in speed offered by the all-moves-as-first heuristic is very important. In contrast to the basic idea or PP, the number of random games to be played becomes independent of the number of legal moves. This is the main feature of this heuristic. Instead of playing a 9x9 game in more than two hours by using the basic idea, OLGA plays in five minutes with the use of this heuristic. However, we have seen two problems due to the use of this heuristic. Therefore, how do the uses of all moves as first heuristic and progressive pruning compare in strength?

Table 3 shows the mean of the relative scores of OLGA(Basic idea) and OLGA(PP) against OLGA(all moves as first).

<table>
<thead>
<tr>
<th>Basic idea</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>+13.7</td>
<td>+4.0</td>
</tr>
</tbody>
</table>

While the previous section underlines that PP decreases the level of OLGA by about five or ten points according to the value of $r_d$, the all-moves-as-first heuristic decreases the level by almost fifteen points. The confrontation between OLGA(PP) and OLGA(all moves as first) shows that PP remains better in strength.

4.2.3 Influence of the Number of Random Games. The standard deviation $\sigma$ of the random games usually amounts to 45 points at the beginning and in the middle game, and diminishes in the endgame. If we play $N$ random games and take the average, the standard deviation is $\sigma/\sqrt{N}$. This calculation helps find how many random games to play so that the evaluations of the moves become sufficiently close to their expected outcome. From a practical point of view the question is: how does this relate to the level of play? Table 4 shows the result of Oleg($N = 10,000$) against OLEG($N = 1000$) and OLEG($N = 100,000$).

<table>
<thead>
<tr>
<th>1000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>-12.7</td>
<td>+3.2</td>
</tr>
</tbody>
</table>

We can conclude that 10,000 random games per move is a good compromise when using the all-moves-as-first heuristic. Since Oleg is able to play 10,000 random games per second, this means it can play one move per second while using only this heuristic.

4.3 Temperature

Instead of making the temperature start high and decrease as we play more random games, it is simpler to make it a constant. The temperature has been
implemented in OLEG in a somewhat different way as in GOBBLE. In the latter, two lists of moves were maintained for both players, and the moves in the random games were played in the order of the lists (if the move in the list is not legal, we just take the next in the list). Between each random game, the lists were sorted according to the current evaluation of the moves and then moves were shifted in the list with a probability depending on the temperature.

In OLEG, in order to choose a move in a random game, we consider all the legal moves and play one of them with a probability proportional to
\[ \exp(Kv), \]
where \( v \) is the current evaluation of the move and \( K \) a constant which must be seen as the inverse of the temperature (\( K = 0 \) means \( T = \infty \)). A drawback of this method is that it slows down the speed of the random games to about 2,000 per second. Table 5 shows the results of OLEG(\( K = 2 \)) against OLEG(\( K \)) for a few values of \( K \).

<table>
<thead>
<tr>
<th>( K )</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>-8.1</td>
<td>+2.6</td>
<td>-4.9</td>
<td>-11.3</td>
</tr>
</tbody>
</table>

Table 5. Relative scores of OLEG with different values of \( K \) against OLEG(\( K = 2 \)).

So, there is indeed something to be gained by using a constant temperature. This is probably because the best moves are played early and thus, obtain a more accurate evaluation. However, it is bad to have \( K \) too large. The best we have found is \( K = 5 \).

4.4 Simulated Annealing

Simulated annealing (Kirkpatrick, Gelatt, and Vecchi, 1983) was presented in Bruegmann (1993) as the main idea of the method. We have seen that it is perfectly possible not to use it, so the question arises: what is its real contribution?

To answer the question we performed some experiments with simulated annealing in OLEG. In our implementation the variable \( K \) increases as more random games are played. However, we have not been able to achieve significantly better results this way than with \( K \) set to a constant. For example, we have made an experiment between OLEG with simulated annealing and \( K \) varying from 0 to 5, and OLEG with \( K = 5 \). The version with simulated annealing won by 1.6 points in average.

The motivation for using simulated annealing was probably that the program would gain some reading ability, but we have not seen any evidence of this, the program making the same kind of tactical blunders. Besides, the way simulated annealing is implemented in GOBBLE is not classical. Simulated annealing normally has an evaluation that depends only on the current state (in the case of GOBBLE, a state is the lists of moves for both players); instead in GOBBLE the evaluation of a state is the average of all the random games that are based on all the states reached so far. There may be a way to design a
true simulated-annealing-based Go program, but speed would, then, be a major concern.

4.4.1 OLEG against VEGOS. VEGOS is a recent go program available on the web (Kaminski, 2003). It is based on the same ideas as GOBBLE; particularly it uses simulated annealing. A confrontation of 20 games against OLEG($K = 2$, without simulated annealing) has resulted in an average win of 7.5 points for OLEG. We did not perform more games because we had to play them by hand. The playing styles of the programs are similar, with slightly different tactical weaknesses. The result of this confrontation is another reason why we doubt that simulated annealing is crucial for Monte-Carlo Go.

4.5 Depth-2 Enhancement

For the depth-2 enhancement the given position is the root of a depth-two min-max tree. Let us start the random games from the root by two given moves, one move for the friendly side, and, then, one move for the opponent, and make statistics on the terminal position evaluation for each node situated at depth 2 in the min-max tree. At depth-one nodes, the value is computed by using the min rule. When a depth-one value has been proved to be inferior to another one, then this move is pruned, and no more random games are started with this move first. This variant is more complex in time because, if $n$ is the number of possible moves, about $n^2$ statistical variables must be sampled, instead of $n$ only.

We set up a match between two versions of OLGA using progressive pruning at the root node. OLGA(Depth=1) backs up the statistics about random games at depth one while OLGA(Depth=2) backs up the statistics at depth two and uses the min rule to obtain the value of depth-one nodes. The values of the parameters of OLGA(Depth=1) are the same as the parameters of the PP program. The minimal number of random games without pruning is set to 100. The maximal number of random games is also fixed to 10,000 multiplied by the number of legal moves, $r_d$ is set to 1, and $\sigma_e$ is set to 0.2. While OLGA(Depth=1) only uses $10^4$ per 9x9 game, OLGA(Depth=2) is very slow. In order to speed up OLGA(Depth=2), we use the all moves as first heuristic. Thus, it uses about 2 hours per 9x9 game, which yields results in a reasonable time.

Table 6 shows the mean of the relative score of Prog(Depth=2) against Prog(Depth=1), Prog being either OLGA or OLEG.

<table>
<thead>
<tr>
<th>OLGA</th>
<th>OLEG</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2.1</td>
<td>-2.4</td>
</tr>
</tbody>
</table>

*Table 6. Relative scores of Prog(Depth=2) against Prog(Depth=1).*

Intuitively, the results should be better for the depth-two programs, but they are actually slightly worse. How can this be explained?

The first possible explanation lies in the min-max oscillation observed at the root node when performing iterative deepening. A depth-one search overestimates...
the min-max value of the root while a depth-two search underestimates the min-max value. Thus, the depth-two min-max value of the root node is more difficult to separate from the evaluation of the root (also obtained with random simulations) than the depth-one min-max value is. In this view, OLGA(Depth=2) pass on some positions on which OLGA(Depth=1) does not. In order to obtain an answer to the validity of this explanation, a depth-three experiment becomes mandatory. If depth three performs well, then the explanation should be reinforced, otherwise another explanation is needed.

The second explanation is statistical. Let \( Z \) be a random variable which is the maximum of 10 identical random variables \( X_i \) (0 ≤ \( i \) ≤ 9) with mean(\( X_i \)) = 0 and standard deviation \( \sigma(X_i) = 1 \), plus a last one \( Y \) with mean(\( Y \)) = \( \delta > 0 \) and standard deviation \( \sigma(Y) = 1 \). We have \( Z = \max(X_0, \ldots, X_9, Y) \). Table 7 provides the mean and standard deviation of \( Z \).

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean(( Z ))</td>
<td>1.58</td>
<td>1.77</td>
<td>2.27</td>
<td>3.06</td>
<td>4.01</td>
</tr>
<tr>
<td>( \sigma( Z ) )</td>
<td>0.58</td>
<td>0.62</td>
<td>0.77</td>
<td>0.92</td>
<td>0.98</td>
</tr>
</tbody>
</table>

Table 7 shows that, on positions in which all 11 moves are equals (\( \delta = 0 \)), performing a max (resp. min) leads to a positive (resp. negative) value (1.58) significantly greater (resp. smaller) than the (resp. opposite of the) standard deviation of each move (1). Therefore, when performing a depth-two search, the depth-one nodes are largely underestimated and, given these depth-one estimations, the root node is largely overestimated. Thus, when the number of games is not sufficient, the error propagates once in the negative direction and then in the positive one. To sum up, when the moves are almost equal, the min-max value at the root node contains a great deal of randomness.

Table 7 also points out another explanation. When \( \delta \leq 2 \), mean(\( Z \)) and \( \sigma(Z) \) remain quite different from \( \delta \) and 1 respectively. But when \( \delta \geq 4 \), both mean(\( Z \)) and \( \sigma(Z) \) are almost equal to \( \delta \) and 1 respectively. Thus, on positions with one best move only and ten average moves, the mean value of the max value becomes exact only when the difference between the best move evaluation and the other move evaluation is about four times the value of the standard deviation of the move evaluations.

These two remarks show that, when using the depth-two enhancement, a great deal of uncertainty is contained in the min value of depth-one nodes and even more in the min-max value of the root node.

### 4.6 An All-against-All Tournament

To evaluate the Monte-Carlo approach against the knowledge-based approach, this subsection provides the results of an all-against-all 9x9 tournament between OLGA, OLEG, INDIGO and GNUGO. GNUGO (Bump, 2003)
Monte-Carlo Go Developments

is a knowledge-based Go program developed by the Free Software Foundation. We used the 3.2 version released in April 2002. INDIGO2002 (Bouzy, 2002) is another knowledge-based program whose move decision process is described in Bouzy (2003). OLGA means OLGA(\textit{Depth}=1, r_\textit{d}=1, \sigma_\textit{c}=0.2) using PP and not the all-moves-as-first heuristic. OLEG uses the all-moves-as-first heuristic, a constant temperature corresponding to \textit{K}=2, and it does not use PP. Table 8 shows the grid of the all against all tournament.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
 & Olga & Indigo & GNUGo \\
\hline
Oleg & +10.4 & -4.9 & +31.5 \\
Olga & +1.8 & +33.7 & +8.7 \\
Indigo & & & \\
\hline
\end{tabular}
\caption{The grid of the all against all tournament.}
\end{table}

First, Monte Carlo excepted, our tests show that, on 9x9 board, GNUGO 3.2 is about 8.7 points better than INDIGO2002. Then, considering Monte Carlo, both OLGA and OLEG are far below GNUGO (more than thirty points average). However, given the very large difference of complexity between the move generator of GNUGO and our move generators, this result is quite satisfactory. Against INDIGO, both OLGA and OLEG perform well. The three programs beat themselves circularly. On 9x9 boards, we may say that OLEG and OLGA containing very little knowledge have a comparable level to the level of INDIGO that contains a large amount of knowledge. The result between two very different architectures, statistical and knowledge, is quite enlightening.

Besides, we have made tests on larger boards. Although the number of games played is not sufficient to obtain significant results, they give an idea of the behaviour of Monte-Carlo programs in such situations. On the basis of twenty 13x13 games only, OLGA is 17 points below INDIGO. On a 19x19 Go board, a 7 games' confrontation between OLEG and GNUGO was won by GNUGO with an average margin of 83 points. OLEG takes a long time to play (about 3 hours per game) for several reasons. First, the random games are longer. Second, we must play more of them to have an accurate evaluation of the moves (we did it with 50,000 random games per move). Lastly, the main game itself is longer. In those games, typically OLEG makes a large connected group in the centre with just sufficient territory to live and GNUGO gets the points on the sides.

5. Discussion

While showing a sample game between OLEG and its author, this section discusses the strengths and weaknesses of the statistical approach and opens up some promising perspectives.

5.1 Strengths and Weaknesses

On the programmer's side, the main strength of the Monte-Carlo approach is that it uses very little knowledge. First, a Monte-Carlo game program can
be developed very quickly. As Bruegmann (1993) did for the game of Go, this upside must be underlined: the programmer has to implement efficiently the rules of the game and eyes, and that is all. He can leave all other knowledge aside. Second, the decomposition of the whole game into sub-games, a feature of knowledge-based programs, is avoided. This decomposition introduces a bias in knowledge-based programs, and Monte-Carlo programs do not suffer from this downside. Finally, the evaluations are performed on terminated games, and, consequently, the evaluation function is trivial. Besides, Monte-Carlo Go programs are weak tactically, and they are still slower than classical programs and, at the moment, it is difficult to make them play on boards larger than 13x13.

In the human user's viewpoint, any Monte-Carlo Go program underestimates the positions for both sides. Thus, it likes to keep its own strength. As a result, it likes to make strongly connected shapes. Conversely, it looks for weaknesses in the opponent position that do not exist. This can be seen in the game of Figure 4. It was played between OLEG as Black and its author as White. OLEG was set with $K = 5$ and 10,000 random games per move. White was playing relatively softly in this game and did not try to crush the program.

![Figure 4. OLEG(B)-Helmstetter(W). White wins by 17 points plus the komi.](image)

### 5.2 Perspectives

First, eliminate the tactical weakness of the Monte-Carlo method with a processing containing tactical search. Second, use domain dependent knowledge to play pseudo-random games. Third, build statistics not only on the global score but on other objects.

#### 5.2.1 Preprocessing with Tactical Search

The main weakness of the Monte-Carlo approach is tactics. Therefore, it is worth adding some tactical modules to the program. As a first step it is easy to add a simple tactical module which reads ladders. This module can be either a preprocessing module
or a post-processing module to the Monte-Carlo method. In this context, each module is independent of the other one, and does not use the strength of the other one. Another idea would consist in making the two modules interact. When the tactical module selects moves for the random games, it would be useful for Monte Carlo to use the already available tactical results. This approach would require a quick access to the tactical results, and would slow down the random games. The validity of the tactical results would depend on the moves already played and it would be difficult to build an accurate mechanism to this end. Nevertheless, this approach looks promising.

5.2.2 Using Domain Dependent Pseudo-random Games. Until now, a program using random games and very little knowledge has a level comparable to INDIGO2002. Thus, what would be the level of a program using domain dependent pseudo-random games? As suggested by Bruegmann (1993), a first experiment would be to make the random program use patterns giving the probability of a move advised by the pattern. The pattern database should be built a priori and should not introduce too much bias into the random games.

5.2.3 Exploring the Locality of Go with Statistics. To date, we have estimated the value of a move by considering only the final scores of the random games where it had been played. Thus, we obtain a global evaluation of the move. This is both a strength and a weakness of the method. Indeed, the effect of a move is often only local, particularly on 19x19 go boards. We would like to know whether and why a move is good.

It might be possible to link the value of a move to more local subgoals from which we could establish statistics. The value of those subgoals could, then, be evaluated by linking them to the final score. Interesting subgoals could deal with capturing strings or connecting strings together.

6. Conclusion

In this paper, we described a Monte-Carlo approach to computer Go. Like Bruegmann's (1993) Monte-Carlo Go, it uses very little domain-dependent knowledge, except concerning eyes. When compared to the knowledge-based approaches, this approach is very easy to implement. However, its weakness lies in the tactics. We have assessed several heuristics by performing experiments with different versions of our programs OLGA and OLEG. Progressive pruning and the all-moves-as-first heuristic enables the programs to play more quickly without decreasing their level much. Then, adding a constant temperature to the approach guarantees a higher level but yields a slightly slower program. Furthermore, we have shown that adding simulated annealing does not help: it makes the program more complicated and slower, and the level is not significantly better. Besides, we have tried to enhance our programs with
a depth-two tree search, which did not work well. Lastly, we have assessed our programs against existing knowledge-based ones, GNUGO and INDIGO, on 9x9 boards. OLGA and OLEG are still clearly inferior to GNUGO (version 3.2) but they match INDIGO.

We believe that, with the help of the ever-increasing power of computers, this approach is promising for computer Go in the future. At least, it provides Go programs with a statistical global search, which is less expensive than global tree search, and which enriches move generation with a kind of verification. In this respect, this approach fills the gap left by global tree search in computer Go (no termination) and left by move generation (no verification). We believe that the statistical search is an alternative to tree search (Junghanns, 1998) worth considering in practice. It has already been considered theoretically within the framework of Rivest (1988). In the near future, we plan to enhance our Monte-Carlo approach in several ways: adding tactics, inserting domain-dependent knowledge into the random games, and exploring the locality of Go with more statistics.

References


STATIC ANALYSIS BY INCREMENTAL COMPUTATION IN GO PROGRAMMING

K. Nakamura
College of Science and Engineering, Tokyo Denki University
Hatoyama-machi, Saitama-ken, 350-0394 Japan.
nakamura@k.dendai.ac.jp

Abstract  Computer-Go programs have high computational costs for static analysis, even though most intersections of the board remain unchanged after one move. Therefore, we introduced the method of incremental computation as an essential feature in Go programming. This paper explores how incremental computation is applied to the static analysis in Go programs, and describes two types of analysis and pattern recognition. One type is determination in cases where the territories of groups are almost determined. This includes (1) the methods of determining the life and death of a group by numerical features and (2) the method of finding the numbers of regions enclosed by the groups based on Euler’s formula. The other type is estimation of groups of stones and territories by analysing the influence of stones using an “electric charge model” in cases where the density of stones is rather low. In the analysis, operations on sets of intersections are used for mathematical descriptions when applying incremental computation as well as definitions of the notions on the Go board.

Keywords: incremental computation, Euler’s formula, life and death, potential distribution, electric charge model

1. Introduction

The strength of computer-Go programs is generally considered as a beginners’ level despite all efforts by many researchers. Many Go players in Japan estimate the current best Go programs as playing at around 4 or 5 kyu in amateur rating, although the Japan Go Association recently certified some Go programs as one dan. This is stronger than 5 kyu; the difference is 5 handicap stones. The progress in playing strength is considered rather slow compared to that of computer Shogi. The latter game is also considered very difficult, but apparently the Shogi programs are steadily improving. We assume that investigating the theoretical and mathematical foundations of the game as well as applying the results in practical Go programming are significant for computer Go.
It is widely accepted that an efficient static analysis is essential to improve the playing strength of computer-Go programs. However, the costs of such an analysis are much higher than those of chess and Shogi. The static analysis needs to be repeated not only at every move, but also at every step in the search tree.

In this paper, we explore how the incremental computation can be applied to static analysis. We discuss two types of static analysis and pattern recognition in computer Go: determination and estimation. The first type, determination, contains the analysis of cases where the territories of the groups have been almost determined. This includes (1) the methods of determining the life and death of a group by the numerical features and (2) the method of finding the numbers of regions enclosed by the groups based on Euler’s formula. The other type, estimation, deals with the estimation of groups of stones and territories on the board when the density of stones is rather low by analysing the influence of stones using an electric charge model.

The aim of the static analysis is to obtain the phase of the board, which is a collection of overall aspects of the board configuration, such as territories of black and white stones, influence of stones, and life and death of the groups. In most cases, the change in board configurations is restricted to one intersection except for capturing, which seldom occurs. The largest part of the phase usually remains unchanged for one move, although there are cases where the phase changes vastly by one move. By using incremental computation for obtaining the phase of the board, we can restrict the evaluation process to the parts changed without repeating the same process for any unchanged part of the configuration. Since the game of Go requires high computational costs for the static analysis, incremental computation is especially effective for computer Go.

In most previous publications on static analysis in computer Go, the main subject dealt with determining the life and death of groups of stones. Those works include: the theoretical study of static life (Benson, 1976); determining the life and death of groups by some local features including perimeters of the empty regions (Chen and Chen, 1999) and by tactical analysis and eye values (Fotland, 2002); and static analysis by position evaluation (Müller, 2002). The application of combinatorial game theory to yose problems (Berlekamp and Wolfe, 1994) is another theoretical result. Nakamura (2000, 2001) presented basic approaches to the life-and-death problem, which included estimating the number of eyes based on Euler’s formula for connected planar graphs and analysing capturing races by semeai graphs.

There are few papers that discuss the method of incremental computation in computer Go so far. Most Go-playing programs seem to have some mechanism for incremental computation. Klinger and Mechner (1996) and Bouzy (1997) describe some methods for incremental updating of data in Go programs. These
two publications contain elements of the basics of incremental computation since they take into account the knowledge maintenance and backtracking.

Since the early program by Zobrist (1969), most Go programs, including INDIGO (Bouzy, 1995), GO INTELLECT (Chen, 1989), HANDTALK (Chen, 2002), EXPLORER (Müller, 2002), and JIMMY 5.0 (Yan and Hsu, 2001) employ mechanisms for evaluating the influence of stones and determining territories. An important feature of our electric charge model is the computation of the potential distribution which is based on incremental computation. Another feature is that some aspects of Go boards can be described in detail by potential distributions.

This paper is organized as follows. In Section 2, we describe operations on the set of intersections on the board, which are used for representing features of pattern analysis as well as mathematical descriptions of incremental computation. Section 3 describes methods of recognizing blocks and groups based on the set operations, and discusses a method of identifying the life and death of a group enclosing a region by the numerical features of the regions defined by the set operations. Section 4 shows an improved method of estimating the number of regions enclosed by the groups based on Euler's formula for planar graphs. Section 5 outlines another approach of static analysis for recognizing groups and finding the influence of stones based on the electric charge model and on incremental computation.

2. Set Operations and Incremental Computation

In this section, we define several constants and some operations on the sets of intersections. We show the relation of the operations with incremental computation. Our intention is not to use the sets of intersections and the operations directly for the analysis, but to define basic notions on Go boards and to use incremental computation only for the parts that changed in every move.

2.1 Operations on Sets of Intersections

The Board is the set $B = \{(i, j) \mid 1 \leq i, j \leq N\}$ of intersections. In the standard rule $N$ is 19. A configuration is represented by two disjoint sets $B \subseteq B$ and $W \subseteq B$ of intersections occupied by black and white stones, respectively. The intersections in $B$ or $W$ are called black or white stones, respectively. The other elements of Board, $B - B - W$, are empty intersections. An intersection $(i, j)$ is adjacent to an intersection $(m, n)$, if and only if $|i - m| + |j - n| = 1$. An intersection $(i, j)$ is adjacent to a set $S$ of intersections, if and only if $(i, j) \not\in S$ and there is $(m, n) \in S$ such that $(i, j)$ is adjacent to $(m, n)$.

The board $B$ and the empty set $\emptyset$ are constants. Another constant is $Edge \triangleq \{(i, j) \mid i = 1, i = N, j = 1 \text{ or } j = N\}$. 
We have three types of operations: Boolean, shift, and extended operations. The Boolean operations include union $\cup$, intersection $\cap$ and set difference $\sim$. There are four shift operations. The operation Shift Left $\rightarrow A$ is defined by $\rightarrow A = \{(i-1, j) | (i, j) \in A, i \geq 2\}$. The value of $\rightarrow A$ is the set of intersections which are shifted left from the intersections in $A$. The intersections on the left edge in $A$ are eliminated. Other shift operations are: Shift Right $\leftarrow A$, Shift Down $\downarrow A$, and Shift Up $\uparrow A$; they are defined analogously.

For a set $X$, it holds that $|X|$ is the number of elements in $S$. The following extended operations are used for representing features of enclosed regions in Subsection 3.3.

$$exterior(X) \triangleq \{(i, j) | (i, j) \text{ is adjacent to } X\}$$

$$thicken(X) \triangleq X \cup exterior(X)$$

$$\#adjacent(X) \triangleq |X \cap \overline{X}| + |X \cap X \downarrow|$$

Some examples of these operations are shown in Figure 1. We represent a configuration (Figure 1(a)) by sets $B$ (Figure 1(b)) and $W$ (Figure 1(c)) of black and white stones, respectively. The value of $\#adjacent(B)$ is 11, and that of $\#adjacent(W)$ is 3.

### 2.2 Operations and Incremental Computation

Let $Y$ be any set of stones of the same colour, and $A$ be a set of one stone of the same colour, such that $Y \cap A = \emptyset$. Incremental computation of an operation $Op$ for $Y \cup A$ means finding the result $Op(Y \cup A)$ from the value
Static Analysis by Incremental Computation in Go Programming

Op(Y) and the operations on the neighbour intersections of A. The costs of incremental computation are generally lower than those of a full computation, since the change caused by adding a stone in A is restricted to the neighbour intersections of this stone. The results of incremental computation for the basic operations are as follows.

\[ X \cup (Y \cup A) = (X \cup Y) \cup A \]
\[ X \cap (Y \cup A) = (X \cap Y) \cup (X \cap A) \]
\[ X - (Y \cup A) = (X - Y) - A \]
\[ (Y \cup A) - X = (Y - X) \cup (A - X) \]
\[ \overline{Y \cup A} = \overline{Y} \cup \overline{A} \]

Incremental computation for other shift operations is defined analogously. We note that the rightmost terms in the equations represent the changes. We also note that \( A - X = \emptyset \), if \( A \subset X \), and otherwise \( A - X = A \). The results of the method for the extended operations are shown below, with \( |S| \) being the number of elements in a set \( S \).

\[ \text{thicken}(Y \cup A) = \text{thicken}(Y) \cup \text{thicken}(A) \]
\[ \text{exterior}(Y \cup A) = \text{thicken}(Y) \cup \text{thicken}(A) - (Y \cup A) \]
\[ = (\text{exterior}(Y) - A) \cup (\text{exterior}(A) - Y) \]
\[ \#\text{adjacent}(Y \cup A) \]
\[ = |(Y \cup A) \cap (\overline{Y} \cup \overline{A})| + |(Y \cup A) \cap (Y \downarrow \cup A \downarrow)| \]
\[ = \#\text{adjacent}(Y) + |Y \cap \text{exterior}(A)| \]

3. Static Analysis Based on Set Operations

In this section, we define blocks and groups, and discuss a method of determining the life and death of a group that depends on the shape of the enclosed region and on the positions of the opponent stones in the region. We implemented and tested most of the methods in Sections 3 and 4 in Prolog.

3.1 Blocks and Group

A connected set of intersections is defined by the following recursive rules.

1. A set of one intersection is connected.
2. For any set of \( T \) of intersections, if a subset \( S \subseteq T \) is connected, then \( \text{thicken}(S) \cap T \) is connected.

We represent a board configuration by sets \( B \) and \( W \) of black and white stones. The set \( E \) of empty intersections is given by \( E = B - B - W \). A black block is a connected set \( B_X \subseteq B \) such that \( \text{thicken}(B_X) \cap B = B_X \). White
blocks are defined analogously. An empty region is a connected set $E_X \subseteq E$ such that $\text{thicken}(E_X) \cap E = E_X$.

A liberty, or dame, of a black (or white) block $B_X (W_X)$ is an empty intersection in the exterior of the block. Hence we have

$$\text{liberty}(B_X, W) \triangleq \text{exterior}(B_X) \cap E = \text{exterior}(B_X) - W.$$  

We note that every block has non-empty liberties, since any block without the liberty is dead and removed from the board.

The configuration in Figure 1 (a) contains two black blocks, two white blocks and three small empty regions. The inner black block has two liberties. The two black blocks enclose the region of five white stones and five empty intersections.

A group is an important notion that is defined to be either a block or a union of blocks of the same colour such that the blocks are “dynamically” connected, i.e., the opponent cannot cut, or separate, the blocks. Although some groups, such as blocks connected by kosumi (diagonal) relations, can be recognized by static analysis, precise recognition needs dynamic analysis, as discussed in Nakamura (2002), We call the group in this narrow sense the linked group, which is a set of stones connected by adjacent-to or kosumi relations. In Section 5 it is shown that most of the groups in the broad sense are recognized by static analysis based on the electric charge model.

A group is alive, if the opponent player cannot capture it, and dead otherwise. Practically, a group is alive, if it has two eyes (i.e., small enclosed regions), it can be changed to form two eyes or a seki, or the group side wins the capturing race relating this group. There is a case where the life and death depends on a ko in the group.

### 3.2 Life and Death of Groups Enclosing Regions

Following Berlekamp and Wolfe (1994) and Chen and Chen (1999), we represent the types of enclosed regions related to the life and death of groups by pairs $\{\alpha|\beta\}$ of symbols, where $\alpha$ represents the state, if the group side moves next, and $\beta$, if the opponent moves next. The symbol of the state is either $L$, $O$, $S$, or $K$. Symbol $L$ denotes that the group is alive in the sense that the enclosed region can form two eyes, whereas $S$ denotes that the group enclosing the region can form a seki. Although the group is alive in the both cases, we distinguish $S$ from $L$, because the opponent group in the region is also alive in the seki. Symbol $O$ denotes that the region cannot be two eyes but only one eye. Symbol $K$ denotes that the region can be changed to have a ko such that it can have two eyes, if the group side wins the ko, and one eye otherwise. Possible combinations in this section are $\{L|L\}$, $\{L|O\}$, $\{S|O\}$, $\{O|O\}$, $\{L|K\}$, and $\{K|O\}$. In some cases, life and death depends on the outer liberties of the group as well as the features of the region. Note that $\{L|L\}$ corresponds to 2.0 eyes, $\{L|O\}$ 1.5 eyes, and $\{O|O\}$ 1.0 eye in other
static analysis by incremental computation in Go programming

<table>
<thead>
<tr>
<th>Feature</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of region $R$</td>
<td>$</td>
</tr>
<tr>
<td>perimeter of $R$</td>
<td>$</td>
</tr>
<tr>
<td>num. of adjacent-to relations in $X$</td>
<td>$#\text{adjacent}(X)$</td>
</tr>
<tr>
<td>max. neighbours in $X$</td>
<td>$\text{max}_\text{neighbour}(X)$</td>
</tr>
<tr>
<td>num. of opponent stones in $R$</td>
<td>$</td>
</tr>
<tr>
<td>max. liberties of one stone in $B \cap R$</td>
<td>$\text{max}_\text{liberties}(B \cap R)$</td>
</tr>
<tr>
<td>total num. of opponent stones in $R$</td>
<td>$\text{exterior}(R \cap B)$</td>
</tr>
<tr>
<td>num. of intersections in $R$ on the edge</td>
<td>$</td>
</tr>
<tr>
<td>outer liberties of the group</td>
<td>$</td>
</tr>
</tbody>
</table>

Table 1. Features for determining the life and death of a group enclosing a region.

Publications (Chen and Chen, 1999; Fotland, 2002). Since we discuss only the states of closed regions enclosed by groups and exclude the case where the region contains an opponent group with two eyes, we do not use the symbols for the states of half eyes or empty eyes.

Table 1 shows the list of features used for determining the life and death of the groups. In this table, $R$ denotes the region, i.e., the set of intersections enclosed by a group, $E$ the set of empty intersections, and $B$ the set of opponent stones. We assume that the white group encloses the region in the figures. This table contains two features defined as follows.

$$\text{max}_\text{neighbour}(X) \triangleq \max_{p \in X} |X \cap \text{exterior}({p})|$$

$$\text{max}_\text{liberties}(X) \triangleq \max_{p \in E} |E \cap \text{exterior}({p})|$$

We tested this set of features for various patterns of the enclosed regions, and found that the features are effective to identify the types of life and death of the regions with the size of five to eight including those in the corners and those containing opponent stones.

Chen and Chen (1999) have shown that the life and death of a group enclosing an empty region $R$ can be determined by the features, the perimeters of $R$, and the existence of square $$, which is given by $\#\text{adjacent}(R) - |R| + 1 \geq 1$ in our terminology. Another possible set of features for this recognition is $|R|$, $\#\text{adjacent}(R)$, and $\text{max}_\text{neighbour}(R)$.

Figure 2 shows typical empty regions in the corner with $\{L|K\}$ or $\{L|S\}$, if the groups have a few outer liberties. For example, the bent four in the corner (a) is in $\{L|L\}$, if the white group has two or more outer liberties, and $\{L|K\}$ otherwise. White can choose $\{L|K\}$ or $\{L|S\}$ in (e), if the outer liberties are zero or one. These patterns can be identified by the features, the number of intersections on the edge, the perimeters of $R$, and the number of squares, $\#\text{adjacent}(R) - |R| + 1$. 


Figure 2. Patterns of groups with \( \{L|K\} \) and/or \( \{L|S\} \) when the white groups have a few outer liberties.

Figure 3 shows examples of patterns of the enclosed regions containing two or three opponent stones (prisoners). We can identify each of the patterns by the five features in the table. Note that the groups enclosing the patterns (c) and (e) are alive by squashing (oshitsubushi), if the group has one or more outer liberties, otherwise the group is in seki.

A characteristic of our method is that the recognition is based on numerical features of the regions and groups, to which incremental computation can be applied. The method does not use pattern matching as used in many Go programs, which we consider inefficient and inappropriate for incremental computation.

The method shown in this section is only applicable to the groups enclosing closed regions. To analyse patterns with incompletely closed regions, or patterns with half eyes or open eyes, several methods have been proposed such as those by eye values and eye regions in Chen and Chen (1999) and Fotland (2002) and by position evaluation in Chen (2002). We are working on extending our methodology so that it can be applied to incompletely closed regions or loosely connected groups, e.g., those connected by bamboo joints.
4. Finding the Number of Enclosed Regions Based on Euler’s Formula

The regions enclosed by groups are important for deciding the life and death of the group, since the eyes are small enclosed regions and a group enclosing a region can be alive as discussed in the previous section. Nakamura (2000, 2002) proposed a method of using a formula to find the number regions enclosed by the groups. In this section, we show an improved method of finding the number of enclosed regions based on the method of incremental computation. The term “group” in this section refers to the linked group.

For any connected planar graph, the number \( N \) of regions enclosed by edges, or minimal loops, is given by Euler’s formula \( N = n - k + 1 \), where \( n \) and \( k \) are the numbers of edges and vertices, respectively. This formula has been applied in computer graphics to find the number of enclosed open regions in digital figures, which are represented by bit arrays (Gray, 1971). Euler’s formula is also applied in finding “holes” in the game Lines of Action (LoA) (Winands, Uiterwijk, and Van den Herik, 2001).

4.1 Application of Euler’s Formula to Go

For the application of Euler’s formula to graphs to find the number of enclosed regions in Go, we consider each stone in a group as a vertex, and each “link” between the stones as an edge. The link is either the “adjacent-to” relation or the diagonal relation of two stones in the group.

\[
\#\text{link}(G) \triangleq \#\text{adjacent}(G) + |G \cap \overrightarrow{G}{}| + |G \cap \overleftarrow{G}{}|
\]

It is remarked that we assume that every stone in a group is connected to at least an other stone in the group by the link.

The group may contain closed loops of stones composed of three stones and three links, e.g., \( \bullet\bullet\bullet \) and \( \bullet\bullet\bullet \). To find the number of enclosed regions (or the number of open loops), the number of the closed loops \( \#\text{closed}\_\text{loop}(G) \) should be subtracted from the number of the minimal loops. The number of regions enclosed by the group \( G \), is given by

\[
\#\text{empty}\_\text{region}(G) \triangleq \#\text{link}(G) - |G| - \#\text{closed}\_\text{loop}(G) + 1.
\]

A group may contain a closed loop of the form \( \bullet\circ\bullet \), which contains two diagonal links. In this case, only one of the diagonal links is valid, since Euler’s formula applies only to planar graphs. For example, the black group in Figure 4 has 16 links including 8 diagonal links, 12 stones and 4 closed loop. The number of enclosed regions is calculated as \( 16 - 12 - 4 + 1 = 1 \). When the intersection A is occupied by a black stone, the numbers of links, stones and
Figure 4. Black groups enclosing one region (a) and two regions (b).

Figure 5. A group in the corner enclosing one region (a) and two regions (b).

closed loops increase by 3, 1 and 1, respectively, and the number of enclosed regions changes to $19 - 13 - 5 + 1 = 2$.

To apply this method to the groups in the peripherals and corners, we consider that there are links between stones on the edge of the board and a special virtual stone called *earth* as shown in Figure 5. To find the number of the virtual links, we first assign the set of stones on the edge $G \cap \square$ to a variable $X$. The number of virtual links is $|X|$, and the number of closed loops with the earth is $|X \cap X| + |X\nrightarrow X|$. We say that the group $G$ is *earthed*, if $X \neq \emptyset$. Since the group in Figure 5 (a) has 11 links including 3 virtual links, 8 stones including earth and 3 closed loops, the number of open loops is $N = 11 - 8 - 3 + 1 = 1$. After placing a black stone at $A$, the number changes to $N = 13 - 9 - 3 + 1 = 2$.

### 4.2 Incremental Computation

For an effective incremental computation of the number of enclosed regions, we consider the change in number caused by placing a stone on an empty intersection $p$ close to a black group $G$, i.e., there is a stone $q \in G$ such that there is a link between $p$ and $q$. For the intersection $p = (i, j)$ not on the edge, let $C(p)$ be the circular sequence of eight neighbour states,

$$S_{i,j+1}, S_{i+1,j+i}, S_{i+1,j}, S_{i+1,j-1}, S_{i,j-1}, S_{i-1,j-1}, S_{i-1,j}, S_{i-1,j+1},$$

where each state $S_{x,y}$ is empty, or a black or white stone around the intersection $p$. The change in the number of regions caused by placing a black stone at $p$
equals $E(p) - 1$, where $E(p)$ is the number of the consecutive subsequences in $C(p)$ satisfying the following conditions.

1. Each state is either empty or a white (opponent) stone.
2. Each subsequence contains one or more elements in $\text{exterior}\{p\}$.

The fact that the change equals $E(p) - 1$ is derived from $E(p) = n' - 1 - L'$, where $n'$ is the change in the number of links and $L'$ is the change in the number of the closed loops caused by adding the stone. Figure 6 shows typical state patterns of neighbours and the increments of the number of regions enclosed by a black group. The symbol $o$ denotes either an empty or a white stone. Note that for the intersection $A$ in Figure 4 (a), the change is one, since $E(A) = 2$. Note also that the intersections $p$ with $E(p) \geq 2$ are considered the vital points.

For the intersection $p$ on the bottom edge, the number $E(p)$ is defined as the number of consecutive subsequence in the sequence,

$$S_{i-1,j}, S_{i-1,j+1}, S_{i,j+1}, S_{i+1,j+i}, S_{i+1,j}.$$ 

The number $E(p)$ is similarly defined for other edges with different directions, for the corners, and for the white stones. The change in the number of regions is $E(p) - 1$, if the group is earthed, and left and right neighbour intersections are empty. Otherwise, the change in the number of regions is $E(p) - 2$. Note that since a stone is placed on the edge in this case, the group changes in being earthed. Figure 7 shows typical patterns of neighbours and their changes in the number of regions. The pattern (c) represents that the change is one, if the group is earthed, and zero otherwise. Patterns (e) and (f) represent two cases in the corner intersection. Since the point A in Figure 5 (a) matches the pattern (c) and the group is earthed, the change in the number of regions is one.

### 4.3 Problems Related to Incremental Computation

The method shown in this section only provides the number of enclosed regions, but no information on the position of the regions, which are necessary for the analysis as performed in Section 3. A practical method for determining the position is using the potential distribution to be described in Section 5.

Moreover, the enclosed regions counted by the method might include false eyes. A false eye occurs, when two blocks are connected by two diagonal links. Hence, a region with one empty intersection enclosed by two blocks is
not a false eye, if the blocks are connected by three or more diagonal links and virtual links (Figure 8). We can calculate the number of the connecting links by subtracting the total number of diagonal links in the two blocks from the number of diagonal links and virtual links in the group. Note that this rule is also effective for determining the life and death of the dragon with two heads, i.e., a group with two blocks connected by two false eyes. Although the condition for unconditional life by Benson (1976) covers these groups, our rule is simpler and appropriate for incremental computation.

Although most other enclosed regions can form one or two eyes as discussed in Section 3, there is the case that a large enclosed region containing an opponent group might form no eye and/or a seki. This problem can be solved by analysing capturing races (Nakamura, 2001).

5. Static Analysis by Electric Charge Model

This section outlines how incremental computation is used in estimating groups and territories based on the electric charge model. For a board configuration, the potential of each intersection is defined as follows.

1. Each stone distributes potential values $1/d$ to intersections around this stone, where $d$ is the Manhattan distance between the stone and the intersection. The potential of every intersection is the sum of the potential values given by all the stones nearby. The potential given by black stones and that by white stones are separately calculated.

2. Stones close to the edges or the corners have their mirror images as shown in Figure 9. Therefore, the intersections near the edges or the corners have higher potential than intersections in the centre of the board.
A potential value of an intersection given by a stone is reduced, if the intersection is in the shadow of another stone. The distance $d$ is increased by the value shown in Figure 10 in the calculation of the potential value.

We use two different types of potential distributions. One is the potential distribution reflecting only the shadows of the stones of the same color and the other is the potential distribution reflecting the shadows of all stones. In Subsection 5.3 the latter type of distribution is used for recognizing groups. The first type is intended to be used for estimating the strength of the groups, although the use is not shown in this paper.

Figure 11 shows examples of potential distributions. Because of the mirror images, the intersections in the corner have higher potentials as shown in Figure 11(c). In general, the potential of an intersection represents the degree to what extent the intersection is surrounded by stones. The potential in an enclosed region is approximately 4 to 6 as shown in Figure 11(b), and independent of the size and the shape of the region or of the stones of the opponent.

### 5.1 Incremental Computation of Potentials

Because of the linear, additive nature of the potential, incremental computation is generally simple, although mutual interactions of the shadows make the computation more complex for the configurations with many stones. We employ the following approximation method for computing a potential distribution.
We restrict the area to which the potential values from a stone are distributed to the set $D$ of intersections in such a way that the distance from the stone to the other intersections is fewer than 8. The stones have their mirror images, only if the distance between the stones and the edge is fewer than 5.

Whenever a stone is placed on an intersection:

(a) the potential of every intersection in the area $D$ is increased by the value $1/d$, where $d$ is the distance between the stone and the intersection; and

(b) for each stone in the area $D$, the decrements by the effect of shadows are subtracted from the potentials of intersections in the two symmetric shadows of the two stones.

Note that the potential at an intersection obtained by the computation method above is slightly different from the one given by the definition in the previous subsection, if the intersection is in double shadows. The errors by the approximation, however, are negligible.
Figure 12 shows a graph of computation time of the potential distributions for each move. For the experiment, we used a Pentium III processor with 1G Hz clock running a program written in C++. The computation results include four potential distributions, i.e., those for two types of distributions for both black and white stones. Although the time increases with the number of stones while the number is fewer than approximately 50, the time is almost constant otherwise.

5.2 Recognition of Groups and Territories by Potential Distributions

Let $B(i, j)$ be the potential of an intersection $(i, j)$ given by black stones, and $W(i, j)$ by white stones. We use the potential distribution that has the effects of shadows by both black and white stones. The procedure for recognizing black groups is as follows.

1. First, select group points from a given configuration by the following rules.
   (a) The intersection occupied by a black stone is a black group point.
   (b) An empty intersection is a black group point, if $B(i, j) \geq v_1$ and $B(i, j) - W(i, j) \geq v_2$, where $v_1$ and $v_2$ are parameters.

   The white group points are selected similarly. Based on many experiments, we determined the parameters as $v_1 = 1.0, v_2 = 0.55$.

2. Determine connected sets of group points of the same colour (Note that this process is similar to that for determining blocks). The set of stones in each of the connected sets is a group. The connected set of group points represents the influence range, or the territory, of the group.
In some cases, groups connected by diagonal, or kosumi, relations are not recognized only by the rules above. Hence, it is necessary to unify these groups into one by finding stones with this relation.

By testing this method for various configurations in games by professional players, we found that incremental computation can correctly recognize most groups for the wide range of configurations with more than 20 stones. Figure 13 shows an example of groups in a game (Black: C. Chou and White: M. Takemiya, 1994) evaluated by incremental computation. The dark gray areas represent the white territories, and light gray areas the black territories.

It is generally not difficult to compute the difference in finding the groups, since a group usually expands in each move and the changes of the group points are restricted to intersections near to the point of the move. There is, however, the case that a group is cut and separated by erroneous move(s) or ko threats. This case will be investigated in future research.

5.3 Comparison with Other Approaches

Most Go programs employ some methods of evaluating influence of stones and/or finding territories, including the potential distribution (Zobrist, 1969), the 5/21 algorithm (Bouzy, 1995), and the heuristic territory evaluation by Müller (2002). A feature of our approach is that each stone distributes the potential of $1/d$ to the neighbour intersections for the distance $d$. This methodology is common to those in several Go programs including HANDTALK (Chen, 2002), GO-INTELLECT (Chen, 1989), and JIMMY 5.0 (Yan and Hsu, 2001) in the
sense that each stone distributes, or radiates, some inference values to neighbour intersections. In contrast, these programs employ different methods for calculating the values except HANDTALK, in which the distribution of values is similar to our distribution led by $1/d$. For example, GO-INTELLECT uses the exponential function ($\exp(-d)$) instead of $1/d$.

A unique feature of our method in addition to incremental computation is that the influences of black stones and white stones are separately calculated. This is different from most other programs, in which the influence values by black (or white) stones are subtracted from those by white (black) stones to form a single distribution of influence values.

Another unique feature of our method is that it uses the mirror images, the shadows, and four kinds of potential distributions to describe some aspects of Go boards in detail. By these features, the potential values of every intersection in the board represent how strong other black and white stones surround the intersection. Note that this property is based on the potential given by $1/d$ and the mirror images.

6. Concluding Remarks

In this paper, we discussed static analysis based on incremental computation to be used in the static analysis in Go programming. The main questions were: (1) how the incremental computation can be applied to the static analysis, (2) how much does the computation speed increase by incremental computation, and (3) what sort of analysis is suitable or unsuitable for incremental computation? We showed applications of our method to static analysis in Go programming, including:

- identifying the life and death of a group enclosing a region by numerical features, which are described by the operations on sets of intersections;
- finding the number of regions enclosed by a group based on Euler’s Formula; and
- estimation of groups and territories by potential distributions based on the electric charge model.

The analysis methods are based on numerical features or values, and not on pattern matching. Most notions in the static analysis and incremental computation are mathematically defined by the operations on sets of intersections. We showed that incremental computation can be used for the operations in the analysis.

The author and his colleagues are implementing the methods described above in a Go-playing program in Prolog and C++. There is still some work to be done before we can satisfactorily answer the questions above. Future problems include:

- finding numerical features effective for identifying alive-and-dead patterns of loosely connected groups, especially those in the corners;
K. Nakamura

- developing a method of acquiring a broad class of alive-and-dead patterns and making a database efficiently; and
- developing faster algorithms for the analysis, especially for incremental computation of the potential distributions.

Acknowledgements

The author would like to thank the anonymous referees for their helpful comments. He also would like to thank Shuhei Kitoma, Tomomi Miyashita, Hiroyuki Otsuka, and Ayumi Kondo for their help in implementing the ideas and preparing the manuscript.

References

BUILDING THE CHECKERS 10-PIECE ENDGAME DATABASES

J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, S. Sutphen
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8
{jonathan, yngvi, burch, lake, paullu, steve}@cs.ualberta.ca, http://www.cs.ualberta.ca/

Abstract In 1993, the CHINOOK team completed the computation of the 2 through 8-piece checkers endgame databases, consisting of roughly 444 billion positions. Until recently, nobody had attempted to extend this work. In November 2001, we began an effort to compute the 9- and 10-piece databases. By June 2003, the entire 9-piece database and the 5-piece versus 5-piece portion of the 10-piece database were completed. The result is a 13 trillion position database, compressed into 148 GB of data organized for real-time decompression. This represents the largest endgame database initiative yet attempted. The results obtained from these computations are being used to aid an attempt to weakly solve the game. This paper describes our experiences working on building large endgame databases.

Keywords: Retrograde analysis, endgame databases, checkers

1. Introduction

Endgame databases have had an enormous impact in computer games research. They have been instrumental in building world championship programs (e.g., the World Man-Machine Checkers Champion CHINOOK (Schaeffer, 1997)), solving games (e.g., Nine Men's Morris (Gasser, 1996) and Awari (Romein and Bal, 2002, 2003)), and uncovering new insights into games.

For converging games, where the number of pieces on the board reduces as the game progresses, larger endgame databases are a performance asset to a game-playing program, both in terms of reducing the size of the search tree and by replacing heuristic evaluations with perfect knowledge. However, there are practical considerations to building large databases, including the time required to compute them, and the resulting size of the (compressed) databases. Few researchers and developers have the expertise, motivation, patience, and computing resources to push database technology to its limit (a recent exception is the solution to the game of Awari (Romein and Bal, 2002, 2003)). This means, for example, that the 6-piece chess endgame databases are unlikely to be completed in the near future.
CHINOOK is the World Man-Machine Checkers Champion (Schaeffer, 1997). The 8-piece endgame databases were a critical part of the program's success against the top human players. The databases contained secrets that were well beyond the understanding of even the premier players in the world. These databases were started in 1989 and completed in 1993 — 444 billion positions compressed into 5.6 GB of data. These numbers may seem small by today's standards, but were impressive back in the early 1990s when a state-of-the-art CPU was an Intel 486, 32 MB was considered to be a lot of memory, and 1 GB disks were new technology and very expensive.

Beginning in November 2001, we started production runs for computing the 9- and 10-piece checkers endgame databases. The databases are not needed to improve the playing strength of checkers programs; there are currently at least five checkers programs that are superior to all human players. Rather, there is a more enticing goal: solving the game of checkers (or, more precisely, weakly solving the game (Allis, 1994)). The total search space for the game is $5 \times 10^{20}$, a seemingly prohibitively large number. However, most of the search space is likely to be irrelevant to the proof, and resulting estimates of the proof-tree size are well within what is possible to compute with current technology. Building the 10-piece databases (specifically the key 5-piece versus 5-piece subset, where each side has the same number of pieces) is a key stepping stone to solving checkers.

This paper describes our experiences building the 9- and 10-piece checkers databases. The task was daunting, given the need for 64-bit addressing, large computations (up to 171 billion positions at a time), large intermediate disk needs (over 1 TB), verification of the results, and fault tolerance. In 10 years, these numbers will seem trivial, but the techniques will be useful for the next large database computation.

This paper makes the following contributions:

1. the practical considerations that complicate any long-term data-intensive computation,
2. the system issues that need to be addressed, including memory constraints, concurrency, compression, and fault tolerance,
3. improved data compression techniques,
4. data on the 9- and 10-piece checkers databases, and
5. speculation on the likelihood of solving checkers in the near future.

Section 2 describes the algorithms used to compute the 8-piece databases. Section 3 discusses the enhancements needed to move to the larger 10-piece databases.

---

1There are over 100 checkers variants. The variant used here is played on an $8 \times 8$ board and is popular in the former British Commonwealth and in North America. So-called International Checkers is played on a $10 \times 10$ board and is popular in Russia, Europe, and Africa.
databases. The results from building the databases and the implications for solving the game of checkers are in Section 4. Section 5 concludes with perspectives on building larger databases.

2. **Algorithms**

The important application-specific properties that influence the database algorithms are (Goldenberg et al., 2003) (the “Properties”):

1. The game starts with 12 white and 12 black checkers on the board.
2. A captured piece is removed from the board and cannot return.
3. Checkers can be promoted to become kings (when the checker moves to the back rank of the opponent).
4. Checkers move forward; kings move forward and backward.

The algorithms used for the checkers computation are updated versions of those used to compute the CHINOOK 8-piece databases (Lake et al., 1994). This code had not been touched since the completion of the databases in 1993.

The most common format of an endgame database stores for each position a distance metric. This metric is typically either the number of moves to win (if appropriate) or the number of moves to convert to another database. This level of detail is tremendously useful in practice since it allows a game-playing program to play the “best” database moves without needing any search. However, this representation requires (at least) a byte of data per position, and the resulting database does not compress well. The philosophy adopted for building checkers databases has been to build the largest databases possible. To do this necessitates storing the minimal amount of information per position in the database — recording only whether a position is a win, a loss or a draw. The result facilitates the creation of large endgame databases that compress extremely well.

For database calculations, each position is represented by 2 bits, representing the values win (W), loss (L), at least a draw (D), and unknown (U). Using D to mean at-least-a-draw instead of exactly a draw is useful, since it reduces the amount of disk I/O done by the program (see the Lookups phase described below). A portion of the endgame database (a slice) is computed by resolving all positions as wins, losses or draws. The final result is compressed, verified, and then added to the master copy of the completed databases.

The 10-piece databases are huge (8.5 trillion positions for just the 5-piece versus 5-piece subset), and it is not practical to do the entire calculation as one big computation. Instead, the problem is broken down into smaller slices that can be solved more easily. The databases are broken down as follows:

- By pieces: The N-piece database can be computed once the N-1-piece database is done (by Property #2).
- By material: An N-piece database is further divided so that subsets with a different number of pieces per side can be computed in parallel (Property
1. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, S. Sutphen #2). For example, in the 9-piece database computation, the 8 pieces versus 1, 7 versus 2, 6 versus 3, and 5 versus 4 subsets can be computed in parallel.

- By number of kings: The material division is further broken down by the number of kings for each side (exploiting Property #3). For example, after 5 kings versus 4 kings have been computed, then the subset 4 kings and 1 checker versus 4 kings can be computed (the one checker might promote, thus the 5 king versus 4 king database must be computed first).

- By leading rank: A sub-database is further sliced into pieces by considering the position of each side's most advanced (leading) checker (from ranks 1 to 7). Positions where the leading checker is on rank \( R \) must be computed before those where the leading checker is on rank \( R - 1 \) (Property #4). For example, in the 4 kings and 1 checker versus 4 kings endgame, all positions where the checker is on the seventh rank must be computed before tackling all positions where the checker is on the sixth rank. For databases where each side has a checker, this technique results in dividing the computation into 49 (not-necessarily-equal) slices, dramatically reducing the size of the biggest computation to be performed.

More details on the decomposition can be found in Lake et al. (1994).

Table 1 shows how the 5-piece versus 5-piece subset of the 10-piece database can be subdivided into smaller pieces. The first column gives the number of kings and checkers for the sub-database using the notation "bk\( \) wk\( \) bc\( \) wc\( \)”, where \( bk \) is the number of black kings, \( wk \) is the number of white kings, \( bc \) is the number of black checkers and \( wc \) is the number of white checkers. The 8.5 trillion positions are divided into 21 subsets based on the number of kings and checkers. The 3223 subset (3 kings and 2 checkers for black; 2 kings and 3 checkers for white) is the largest, with roughly 1.6 trillion positions. This is subdivided into 49 slices based on the leading checker.

The largest slices in the 5 piece versus 5 piece subset of the 10-piece database are shown in Table 2. To specify a slice, we use the notation

<table>
<thead>
<tr>
<th>Database</th>
<th>Total Positions</th>
<th>Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>5500</td>
<td>16,257,084,480</td>
<td>1</td>
</tr>
<tr>
<td>5401</td>
<td>142,249,489,200</td>
<td>7</td>
</tr>
<tr>
<td>5302</td>
<td>247,789,432,800</td>
<td>7</td>
</tr>
<tr>
<td>5203</td>
<td>214,750,841,760</td>
<td>7</td>
</tr>
<tr>
<td>5104</td>
<td>92,565,018,000</td>
<td>7</td>
</tr>
<tr>
<td>5005</td>
<td>15,868,288,800</td>
<td>6</td>
</tr>
<tr>
<td>4411</td>
<td>311,375,610,000</td>
<td>28</td>
</tr>
<tr>
<td>4312</td>
<td>1,085,553,705,600</td>
<td>49</td>
</tr>
<tr>
<td>4213</td>
<td>941,518,468,800</td>
<td>49</td>
</tr>
<tr>
<td>4114</td>
<td>406,152,630,000</td>
<td>49</td>
</tr>
<tr>
<td>4015</td>
<td>69,686,136,000</td>
<td>42</td>
</tr>
<tr>
<td>3322</td>
<td>946,853,107,200</td>
<td>28</td>
</tr>
<tr>
<td>3223</td>
<td>1,643,753,217,600</td>
<td>49</td>
</tr>
<tr>
<td>3124</td>
<td>709,688,460,000</td>
<td>49</td>
</tr>
<tr>
<td>3025</td>
<td>121,877,184,000</td>
<td>42</td>
</tr>
<tr>
<td>2233</td>
<td>714,003,388,800</td>
<td>28</td>
</tr>
<tr>
<td>2134</td>
<td>617,101,500,000</td>
<td>49</td>
</tr>
<tr>
<td>2035</td>
<td>106,080,312,960</td>
<td>42</td>
</tr>
<tr>
<td>1144</td>
<td>133,467,390,552</td>
<td>28</td>
</tr>
<tr>
<td>1045</td>
<td>45,934,129,104</td>
<td>42</td>
</tr>
<tr>
<td>0055</td>
<td>3,956,756,472</td>
<td>21</td>
</tr>
<tr>
<td>Total</td>
<td>8,586,481,972,128</td>
<td>630</td>
</tr>
</tbody>
</table>

*Table 1.* Database slices for 10-piece database (5 versus 5 pieces).
Building the Checkers 10-piece Endgame Databases

Table 2. Largest 10-piece database slices.

<table>
<thead>
<tr>
<th>Slice</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>3223.77/2332.77</td>
<td>85,515,674,400 x 2 = 171,031,348,800</td>
</tr>
<tr>
<td>2233.76/2332.67</td>
<td>73,228,209,600 x 2 = 146,456,419,200</td>
</tr>
<tr>
<td>3223.67/2332.76</td>
<td>71,823,866,400 x 2 = 143,647,732,800</td>
</tr>
<tr>
<td>3223.76/2332.67</td>
<td>59,656,240,800 x 2 = 119,312,481,600</td>
</tr>
<tr>
<td>3322.76/3322.67</td>
<td>58,741,300,800 x 2 = 117,482,601,600</td>
</tr>
<tr>
<td>3223.57/2332.75</td>
<td>58,132,058,400 x 2 = 116,264,116,800</td>
</tr>
<tr>
<td>2134.77/2143.77</td>
<td>56,491,266,000 x 2 = 112,982,532,000</td>
</tr>
<tr>
<td>2233.77</td>
<td>104,558,625,600 = 104,558,625,600</td>
</tr>
<tr>
<td>3223.66/2332.66</td>
<td>50,304,477,600 x 2 = 100,608,955,200</td>
</tr>
</tbody>
</table>

"bk wk bc wc br wr" where br is the rank of the leading black checker and wr is the rank of the leading white checker. The largest slice is 171 billion positions (3223.77 with black to move and its mirror database 2332.77 with white to move). Using 2 bits per position, this slice requires almost 40 GB of storage during its computation phase. In total, there are only 9 slices that have a size of over 100 billion positions.

Note that slices can be further sub-divided. Gil Dodgen and Ed Trice (2002) have experimented with using both the rank of the leading checker and the configuration of checkers on the first rank to achieve further subdivisions. The finer granularity of the slices reduces the RAM needs and increases the computation's concurrency. For the work reported here, additional subdivisions were not needed. However, with current technology they might be needed if one wanted to compute the 11-piece databases (currently not in our plans).

The endgame database solving programs were designed with the following objectives in mind: reduce the amount of disk I/O needed, reduce the memory requirements for the largest jobs, and use as many machines as possible. The computation of a database slice consists of 5 phases. The phases iterate over the data, where each position value in the slice has been initialized to unknown (U). The database construction phases are summarized in Table 3.

1 Captures: The rules of checkers require that a capture move, if present in a position, must be played. A capture move removes one or more pieces from the board. All capture moves are looked up in previously computed databases and the maximum of the resulting values (W/L/D) is assigned to the position. For an N-piece database calculation, this phase only requires the 2 through N-1-piece databases. This is important because the N-1-piece databases are considerably smaller than the N-piece databases. For example, the 9-piece databases are only 18 GB in size. Thus the capture phase for all 10-piece database calculations can be computed well in advance of when the data is needed.

2 Lookups: The databases are sliced according to the leading checker. When the leading checker advances, it will result in a position that has
already been computed. The Lookups phase resolves all moves by the leading checker. By handling this I/O in a separate phase, we can guarantee that the next phase (non-captures) does not have to access any previously computed database results.

The advance of the checker may result in the current position being resolved as a win. In rare cases the only moves possible in a position are those of the leading checker. If all these moves lead to losing positions, then the current position can be resolved as a loss. If the leading checker advances and the resulting position leads to a draw, then we have a lower bound on the value of the position. The position might still be a win (a king move or non-leading checker move could lead to a winning position). Thus, if a leading checker move results in a draw score, this position is marked as a D but with the semantics being that the value is ≥ a draw. For this phase, only the N-piece database is needed (but, as explained below, because of the compression scheme used, the 2 through N-1-piece databases might also be required).

3 Non-captures: The preceding phases resolved all requests for information from previously computed database slices. In the non-captures phase, only moves by kings and non-leading checkers are considered. Hence there is no need to access the previously-computed databases. In contrast to the previous phases, the non-captures phase is compute-intensive.

This phase iterates over all positions in the slice, skipping over capture positions (their values are fixed) and W/L positions (their value cannot change). Only unresolved positions and draw positions are considered; the former to discover whether the position is a W/L/D and the the latter to see if the D can become a W. This phase only resolves wins and losses. When no more changes occur during an iteration, the non-captures phase is complete. Any position that has a U or D value must be a real draw.

This phase may require iterating over the data 100 or more times (the maximum number of ply needed to force a winning position into another database slice). To reduce the cost, the program iterates over all positions until a "small" number of changes occurs in an iteration. The positions that change value are saved in a queue. For subsequent iterations, the only positions whose value can be resolved are those that are a predecessor of a queue position.

4 Compression: The endgame databases are needed in a real-time searching program (such as CHINOOK). Hence the data has to be compressed in a way that supports real-time decompression. The compression scheme used is described in Section 3.3.
Table 3. Database construction summary.

5 **Verification**: Errors are a fact of life in any long-running computation. Since one result depends on another, it is critical that the computations be verified for correctness. There is an easy way to do this: after the non-captures phase, a quick scan of the data can verify if the resulting set of values is internally consistent (self-consistency). This is quick, but does not catch all possible errors. Instead, our verification phase operates not on the 2-bit-per-position representation but on the compressed database. All positions are verified that they are consistent not only within the slice, but also with respect to previously computed data. The latter point dramatically increases the cost of the verification, but can find errors not caught by the fast scheme. Besides, it makes it easier to sleep at night!

The database construction phases are summarized in Table 3. The time column is a generic average that represents the percentage of wall clock time spent in each phase. These numbers can vary significantly depending on the data set used. The verification phase is the most expensive since, in effect, it has to repeat most of the work done in the previous phases.

The breakdown of the computation into multiple phases assists in planning how to effectively acquire and use computing resources. The captures, lookups, and verification phases are I/O bound. These phases need to be run on machines with a minimum of 300 GB of disk storage, and they benefit from the fastest possible disk drives. The non-capture phase is compute bound and should be run on the fastest available processor. This phase is easily parallelized, and the performance scales well to a large number of processors on a shared-memory computer.
3. Moving from Eight to Ten Pieces

This section discusses the issues that had to be addressed to enhance the CHINOOK database calculations to accommodate the larger size of the 10-piece databases.

3.1 64-bit Indices

By subdividing the databases into slices, the original CHINOOK code could get by with using 32-bit numbers for position indices. For the 10-piece databases, the largest individual slice was 104 billion positions (the symmetric database 2233.77), necessitating at least 37 bits for addressing.

The CHINOOK code was converted to use 64-bit indices. By-and-large this was easy to do, but there were some subtleties that were initially overlooked. For example, most C compilers do automatic conversion between 32- and 64-bit numbers (both ways), possibly losing precision (and usually not getting a compiler warning). Another danger was intermediate expression results. Some expressions combined 32- and 64-bit data with implicit data conversions that could lead to errors.

Note that simply converting all numbers to use 64 bits was not an option. The tables used for computing position indices occupy a lot of memory. Using 32-bit numbers wherever possible reduced the memory footprint of the program, freeing up more space for disk caching.

3.2 64-bit File Sizes

When we started the project, support for 64-bit file sizes was not fully integrated in Linux. However, we were fortunate in that the experimental kernels we used fully supported the two routines that we needed: open64 and lseek64. Support for large files has limited other groups wanting to build large databases on Windows' platforms.

3.3 Compression

Many endgame databases associate a distance metric with a database position (the number of moves to win or the number of moves to convert to another database slice). For checkers, this was impractical. Our goal was to build the largest database possible. For this to happen, disk space and the execution overhead of accessing the data could not be a limitation. For example, if a byte was associated with each of the 13 trillion database positions computed, then 13 TB of disk would be needed. Even a generous 10:1 compression ratio would still leave the database size at an awkward 1.3 TB. The large disk size will dramatically slow down database computations since it will be difficult
to achieve spatial and temporal disk locality (this was elegantly addressed for smaller databases by Lincke and Marzetta (2000)).

Allowing only win-loss-draw values in the database enables 5 position values to be encoded in a byte ($3^5 = 243 < 256$). Using this trivial compression would result in 13 trillion positions being encoded into 2.6 TB. This is still too large (and expensive) to be practical. Further data compression is needed.

The data has to be available for use in a real-time search. Hence any compression scheme has to support rapid real-time decompression. The databases were compressed by using two techniques: removing information that can be easily re-computed, and run-length encoding.

Any position where either side to move could result in a capture would have the position result removed from the database (i.e., capture and threatened capture positions). It is easy to re-compute the value of a capture position: play the capture move(s) and look up the resulting position(s) in the database. Removing values for positions where a capture is threatened is more problematic. To re-compute this value, the side to move must try all possible moves and, in some cases, in the resulting position the opponent has a forced capture or there is a threatened capture—all these positions must be looked up in the database. Hence positions with a threatened capture may require an expensive search to resolve. It quickly became clear that with our compression algorithms, simply removing capture position values was not good enough; we had to remove threatened capture positions to make the compressed database size reasonable. Our estimate is that removing threatened capture positions improves the compression by a factor of 4.

All capture and threatened capture positions had their value replaced by the dominant value in the database slice. Then run-length encoding would be used to compress the data. The original CHINOOK algorithm encoded 5 positions into a byte (Lake et al., 1994). That left 13 values for the run-length encoding ($256 - 3^5 = 13$). These values were used to represent runs of the dominant value, for runs of length 10 to 3,200. For example, a database slice might be dominated by wins. The capture and threatened capture positions (typically 75% of the positions) would have their values replaced by a win. Run-length encoding would find many long stretches of wins and encode them into one (or a few) bytes.

The original CHINOOK databases, 444 billion positions (all the 2 through 8-piece databases), were compressed into 5.6 GB. This works out to an average of roughly 77 positions encoded in a byte. This is misleading since the lopsided databases (e.g., 6 pieces versus 2) compress very well (they are almost all wins for the strong side), whereas the even material databases (e.g., 4 pieces versus 4 pieces) have a mix of win, loss and draw values, resulting in poorer (but still good compression). The 4 pieces versus 4 pieces database averaged 22 positions per byte.
For the 10-piece databases, our initial estimates were that the above scheme would result in a final database size of 400 GB. Thus it was important to find a better compression scheme. The new algorithm is based on Huffman coding and consists of the following steps:

1. Replace capture and threatened capture positions with the W/L/D value that continues the current run.

2. Convert the above into a string of \((W/L/D, \text{run.length})\) pairs. There will not be two consecutive runs with the same first value.

3. Predict the value of a run based on the value of the run before the previous run. For example, given runs \((\text{draw}, X)\) and \((\text{loss}, Y)\) we would predict the value of the next run to be draw. The prediction is correct roughly 95% of the time. Now convert the string so that a \((\text{value}, \text{length})\) pair simply becomes \text{length}, preceded by a special \text{miss} symbol if the value is not correctly predicted.

4. If a maximum run length of \(N\) is chosen, we then have \(N - 1\) length symbols, one escape symbol that states that an integer length follows, and one symbol that states that the value of this run is predicted incorrectly. Given the frequencies of these symbols, an optimal length limited prefix free code (length limited Huffman code (Turpin and Moffat, 1995)) can be generated. We use a fixed code generated from the largest database file (a separate code per database file does not improve compression much). Twenty bits was chosen as a reasonable limitation on the length of the bit strings, as a table 1,048,576 entries wide used for decoding seemed reasonable and larger string lengths provided minimal improvements. Given this maximum, empirical testing on the databases showed a number around 10,000 to be the best choice for the maximum run length allowed before escaping to a 32-bit integer description. Increasing the number of symbols overly crowded the space of bit strings available for compression by too much, and decreasing the maximum run length increased the number of escaped symbols by too much.

5. The previous types used to predict the types of the first two runs are set by looking ahead at these two symbols and using the values that will correctly predict them. These values are stored at the front of the compressed bit-string using three bits.

With the new scheme the complete 2-piece through 8-piece databases reduce in size from 5.6 to 2.7 GB, cutting the database in half (averaging out to 155 positions per byte). The complete 9-piece databases is 16.8 GB, an average of 227 positions per byte. The 10-piece databases (5 pieces versus 5 pieces) compress to 125 GB, 65 positions per byte. This represents a substantial improvement over the 22 positions per byte seen for the 4 pieces versus 4 pieces subset of the 8-piece databases.
3.4 Disk I/O

Table 3 shows that the wall clock time is dominated by the I/O-intensive phases. The captures, lookups, and verify phases all sequentially proceed through the data. However, each may result in a (usually small) search to resolve the value of the position by looking up values in previously computed databases. This search is a consequence of the data compression scheme used (which removes the value for any capture and threatened capture position). The alternative was to keep the uncompressed data on disk and use that instead. This was not done because of the possibility of introducing an error; the values based on I/O operations (e.g., capture positions) have not been verified for correctness. Rather than trust unverified data, we preferred the (slower) use of the compressed data.

The capture phase runs quite quickly. Surprisingly, typically over 60% of the positions get resolved in this phase. Each position has slightly more than one legal capture move per position. The remaining positions need to have a lookup performed. These positions average roughly 3 moves by the leading checker(s), each of which has to be looked up. Each of these searches is, on average, considerably more expensive than a simple capture position. Thus, even though the lookup resolves only typically 10-15% of database, it runs slower than the captures phase because of the increased amount of I/O.

Each position has I/O performed on it a maximum of two times. Capture positions are visited only in the captures phase; they are not included in the final compressed database, so no verification has to be done. All the remaining positions may have to have I/O done twice: once to do a lookup of any leading checker moves, and once to verify the position value if there is no threatened capture.

The databases have been organized to increase data locality. Database slices that are likely to lead into one another are located physically close to each other in a database file. As well, the program maintains its own internal disk paging, allowing the program to prioritize the database pages kept in memory. The result is that the program, using 200 MB of page buffers, ends up doing one disk I/O for an average of 500 database position value requests. In other words, the hit rate is 499/500.

I/O could be significantly reduced if the database construction program used slices selectively. Some of the databases are relatively small, and slicing them into 49 pieces incurs a lot of unnecessary overhead. These databases could be constructed as one big computation. For example, the 1045 database has only 45 billion positions—using roughly 10.5 GB. Rather than slicing this piece into 42 slices—each with a lookups phase—the entire database could be done as a single computation. Then the lookups would only be required for part of the
database—where there was a leading checker on the 7th rank. This has not been done.

It may seem that the non-captures phase should require the most computational effort, given that this phase must make repeated passes over the data. Further, some of databases are too large to be resident in RAM, requiring costly disk paging. Fortunately, this was not a problem in our implementation. The non-captures phase was set up so that references to values in other databases (requiring I/O operations) were not needed. The position indexing scheme was organized to facilitate spatial and temporal locality. This allowed a (relatively) small working set of data to be resident in memory during the non-captures phase. This was facilitated by having an internal paging mechanism, allowing the program to take advantage of application-dependent properties to minimize the I/O. On our machines, 200 MB of RAM was allocated for pages. With this, we have been able to complete the non-captures phase on files as large as 25 GB in only a few days.

It is interesting to note that the profile of the database computation has changed significantly since we did this work in the early 1990s. Some parts of the program that were previously I/O bound are now CPU bound (more memory to eliminate costly I/O), while other parts that were CPU bound are now I/O bound (CPU speed has improved more than disk speed). This meant that we had to re-profile the program and use additional optimization techniques.

3.5 Errors

Given that this computation takes many CPU years to run and terabytes of data transferred from and to disk, it is critical that an error not be allowed to creep into the calculation. An error early on in the computation, for example, may result in the entire calculation having to be repeated. For example, in October 2001, Gil Dodgen and Ed Trice calculated the 8-piece databases. We compared the CHINOOK results with theirs and discovered a difference in the 7-piece results (Dodgen and Trice, 2002). It eventually turned out that the CHINOOK databases were wrong (a few thousand positions). However, even with the error the databases still passed all our verification tests! This may seem strange, but it can happen. The computed data can be internally consistent, but wrong. The best way to verify the correctness of the databases is to have them independently computed and then the results compared—as we did with the Dodgen/Trice data.2 Needless to say, we are hoping that this experience is not repeated with our 9- and 10-piece calculations.

---

2We are aware of another effort to compute the 9-piece databases and (apparently) the 10-piece databases. We have made two offers to exchange information with this party so that the correctness of both of our efforts could be verified. The offers have been declined.
During the course of the calculations, we had to contend with a faulty CPU, bad memory, a disk crash, network errors and operator errors. In some cases, these errors were trivial to spot (dead disk), while others proved more subtle (faulty memory chip). Precautions were taken to reduce the likelihood of introducing an error into the computation:

1. All calculations were logged. This was useful if a post-mortem was needed to identify the reason(s) for a computation failure.
2. All data copied over a network was verified. The source and destination files had a cyclic redundancy check (CRC) value computed, and the two had to match. In practice, most copies worked correctly. However, at least once a month the CRC check would fail signaling a copy error.
3. The database files were augmented with a 32-bit CRC number for each block of 1024 bytes. Whenever a disk read (local or over the network) was performed, the data read would be verified for consistency with the CRC number. This enhancement allowed us to find a subtle bug in the program, and occasionally would uncover a read failure that was not reported by the operating system.
4. All data computed—databases in their original and compressed form—were archived to tape. Thus, if a catastrophic event occurred (e.g., an error was discovered in the early part of the computation), we would be able to recover by repairing the faulty data rather than having to recompute it from scratch. The need to retrieve data from tape occurred only once.

Despite all the above precautions, occasionally the computation of a database slice failed to verify, even though the logs showed no record of any error occurring.

Are the databases correct? We do not know, but hope that someone will soon repeat our calculations and confirm our results.

3.6 System Issues

For the checkers computation, keeping many machines 100% busy is a difficult task. It is complicated by the calculation dependencies (some databases must be computed before others), hardware specialization (run I/O-intensive jobs on machines with fast disks; run CPU-intensive jobs on machines with fast processors), and disk management (transferring files; making sure that disks do not fill up). We developed tools that can automate most of the computation dependency and hardware specialization issues (Goldenberg et al., 2003). However, managing the data turned out to be labour intensive and a source of potential errors. We were unable to find or build a usable tool that could properly manage the data file dependencies, taking into account disk space constraints, in such a way as to maximize throughput. This appears to be a very difficult
problem, but one that needs to be solved if data-intensive computations are to be fully automated.

4. Results

This section discusses the results of computing all the 9-piece databases and the 5 pieces versus 5 pieces subset of the 10-piece databases.

4.1 Computation

Table 4 shows the sizes of the databases completed.³ 13.1 trillion positions have been computed. We claim that this is the largest endgame database (in terms of number of positions) yet computed for any game.

The computation took 18 months. The 9-piece calculation began in November 2001 and the 10-piece in January 2002. These computations ended in June 2003. Most of the work was completed on dual-processor AMD machines. The memory used ranged from 1 to 4 GB. Older, slower (800 MHz) computers were used to pre-compute the captures phase of the computation. The lookups, non-captures, and verification phases were done using an average of 3 machines, with an average speed of 1.5 GHz. All phases used both processors to speed up the computation.

We had infrequent access to a 64-processor SGI O3000 (500 MHz) with 32 GB of RAM. The machine was used to run the non-captures phase of many of the largest database slices. The database program was parallelized using POSIX threads so that the range of positions could be equally divided between the processors and computed in parallel. The largest computation (171 billion positions) took 2.3 days of SGI time to resolve. The length of time was due to the relative slowness of the processors (500 MHz) and the number of passes over the data that were required to resolve all the positions.

The total amount of computing done is difficult to estimate given that a varying number of machines were used, with different number of processors, and with differing processor speeds. Normalized to a 1.5 GHz processor, a ballpark estimate is that the complete 2 through 9-piece databases and the 5 versus 5 piece subset of the 10-piece databases required 15 CPU years of computing.

Since a few of the 6 versus 4 piece database slices have been computed (low priority on a single machine), we could actually start computing the 11-piece database (6 versus 5 subset). This computation is roughly 10-fold bigger (117 trillion) than what has already been accomplished. We will not pursue this unless the 10-piece databases are insufficient for solving the game of checkers in a reasonable amount of time.

³Note that some 6 piece versus 4 piece slices have been computed.
Building the Checkers 10-piece Endgame Databases

<table>
<thead>
<tr>
<th>Num Pieces</th>
<th>Pieces/Side</th>
<th>Size</th>
<th>Total Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 – 0</td>
<td>120</td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>2 – 0</td>
<td>3,484</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 – 1</td>
<td>3,488</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3 – 0</td>
<td>65,192</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 – 1</td>
<td>196,032</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4 – 0</td>
<td>883,458</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 – 1</td>
<td>3,546,384</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 – 2</td>
<td>2,662,932</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5 – 0</td>
<td>9,237,424</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 – 1</td>
<td>46,409,320</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 – 2</td>
<td>93,041,488</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6 – 0</td>
<td>77,526,288</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 – 1</td>
<td>467,999,856</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 – 2</td>
<td>1,174,279,692</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 – 3</td>
<td>783,806,128</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7 – 0</td>
<td>536,417,856</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 – 1</td>
<td>3,782,903,904</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 – 2</td>
<td>11,404,950,960</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 – 3</td>
<td>19,055,258,760</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8 – 0</td>
<td>3,118,957,920</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 – 1</td>
<td>25,172,147,520</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 – 2</td>
<td>88,657,111,920</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 – 3</td>
<td>177,982,456,720</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 – 4</td>
<td>111,378,534,401</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9 – 0</td>
<td>15,455,930,880</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 – 1</td>
<td>140,531,639,040</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 – 2</td>
<td>566,442,589,440</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 – 3</td>
<td>1,328,448,083,840</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 – 4</td>
<td>1,997,749,399,776</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10 – 0</td>
<td>65,975,569,920</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9 – 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 – 2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 – 3</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 – 4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 – 5</td>
<td>8,586,481,972,128</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>13,144,833,586,271</td>
</tr>
</tbody>
</table>

Table 4. Databases completed.

4.2 Statistics

Because of the concurrency used in the non-captures phase (2 processors would iterate on a slice in parallel), it is hard to know the exact number of ply required to resolve a slice. There were some slices that needed over 180 iterations to resolve, a lower bound that is probably very close to the actual
number. Consider what this number means. There were slices where over 180 ply were needed before a capture could be forced or the leading checker could safely advance one square. In the latter case, one wonders how many more ply would be needed to win the game once that checker had safely advanced a single square—it could be huge! This gives rise to the speculation that there are 10-piece positions that may require many hundreds of ply to solve. For example, Gil Dodgen and Ed Trice have built a perfect-play 7-piece database, and they report the longest win (against best play) to be 253 ply (127 moves) (Trice and Dodgen, 2003). There must be 10-piece positions that are considerably longer than that.

The previous discussion illustrates the disadvantage of computing only W/L/D values. CHINOOK could reach a 10-piece position and not know how to win it. The search could flounder, not being able to choose between winning moves to find a quick path to victory. The (real) danger is that the program will end up cycling around, not knowing how to make progress (although this has not been seen in practice).

4.3 Solving Checkers

The total possible search space for the game of checkers is $5 \times 10^{20}$ (see Table 5)—a daunting number. But how much of it has to be explored to solve checkers? Three assumptions can be used to get a rough upper bound on the effort required to solve checkers. The following heuristics are used to identify the key search space for the proof tree; parts that are excluded may be needed in the case of proving trivially won positions.

- **Material Balance:** An advantage of 2 or more pieces is huge; equivalent to roughly a rook or more in chess. It seems reasonable to assume that a proof would not have to go through positions with lop-sided material. The useful positions are those where the material balance is even, or one side has a single piece advantage.

- **King Balance:** One side having 3 or more kings than the other rarely occurs in practice. Hence we limit the search space to subsets where the number of kings for each side differs by at most 2.

- **Number of Kings:** Kings only appear on the board later in the game. For example, although it is theoretically possible to have 24 pieces on the board with one of them being a king, this scenario is highly contrived. A reasonable assumption is to limit the number of kings to being 6 when there are 10 or less pieces on the board, 4 with 12 or more pieces, 2 with 14 or more pieces, and zero with 24 or less pieces.

Table 5 shows the results of applying the above assumption. From $O(10^{20})$ the potential search space drops to $O(10^{14})$. Of this, the databases computed
thus far represent roughly 7.5 trillion—5% of the reduced search space. It is too early to know the full impact of the 10-piece databases in the checkers proof.

5. Conclusions

Disks are getting larger and cheaper; terabyte systems are affordable and petabyte systems exist. Moore’s law continues to hold and multi-processor systems are ubiquitous. RAM is inexpensive, and hardware and operating systems are gradually moving to accommodate large memories. In effect, there is no technological limit to pushing database technology to even greater heights. The endgame databases reported here contain over $10^{13}$ data points, a 30-fold increase over what seemed possible a decade ago. High-end technology that is available today could be used to push this to $10^{14}$.

The reason for computing the 10-piece databases was to solve the game of checkers. The databases eliminate the bottom of the search tree. A separate project is building the top of the proof tree, searching forward from the root
towards the databases. When the two search frontiers meet, checkers will be solved. At this point in time, it is too early to tell how soon this will happen.

Acknowledgements

This work benefitted from email interactions with Ed Trice and Gil Dodgen. Their independent calculation of the 8-piece database uncovered an error in the CHINOOK databases. It was better to find this error before starting the 9- and 10-piece calculation than afterwards!

This research was supported by grants from the Natural Sciences and Engineering Council of Canada (NSERC) and Alberta’s Informatics Center of Research Excellence (iCORE), and used resources funded by the Canada Foundation for Innovation (the MACI project). The efforts of the Software Systems and Hardware Support Groups of the Computing Science Department, University of Alberta, are greatly appreciated.

References


THE 7-PIECE PERFECT PLAY LOOKUP DATABASE FOR THE GAME OF CHECKERS

E. Trice, G. Dodgen  
Gothic Chess Federation  
GothicChessInfo@aol.com; GilDodgen@cox.net, http://www.GothicChess.org

Abstract Many research teams and individuals have computed endgame databases for the game of chess which use the distance-to-mate metric, enabling their software to forecast the number of moves remaining until the game is over. This is not the case for the game of checkers. Only one programming team has generated a checkers database capable of announcing the distance to the terminal position. This paper examines the benefits and detriments associated with computing three different types of checkers endgames databases, demonstrates the solutions to the longest wins in the 7-piece checkers database, presents tables of longest wins for positions including all permutations of four pieces and fewer against three pieces and fewer, and offers major improvements to some previously published play.

Keywords: checkers, database, endgame, move to win, perfect play

1. Introduction

It is a widespread misconception that since the rules of the game of checkers are simple, so is the playing of the game. This misperception is not limited to just the general public. Several reputable scientific sources have disseminated inaccurate information regarding the state of computer checkers (Gibson, 1993; Schaeffer, 1997 pp. 101-102).

“Computers became unbeatable in checkers several years ago.” Thomas Hoover, "Intelligent Machines," Omni magazine, 1979, p. 162.

“…an improved model of Samuel’s checkers-playing computer today is virtually unbeatable, even defeating checkers champions foolhardy enough to ‘challenge’ it to a game.” Richard Restak, The Brain, The Last Frontier, 1979, p. 336.

“Although computers had long since been unbeatable at such basic games as checkers…” Clark Whelton, Horizon, February 1978.
"So whereas computers can 'crunch' tic-tac-toe, and even checkers, by looking all the way to the end of the game, they cannot do this with chess." Lynn Steen, "Computer Chess: Mind vs. Machine" *Science News*, November 29, 1975.

On August 29, 1992, World Checkers Champion Dr. Marion Tinsley defeated the world's strongest checkers program, CHINOOK (Schaeffer, 1997 pp. 328-332). The score of their match was 4 wins for Tinsley, 2 wins for CHINOOK, and 33 draws each. Tinsley's four wins disproved the notion that checker programs were "unbeatable".

The CHINOOK team experienced a great deal of success while ascending the competition rungs, which allowed them to challenge Dr. Tinsley for the title of "Man vs. Machine" World Champion in 1992. One of the key factors that made CHINOOK such a strong program was the size of its endgame databases. The program eventually had access to 443,748,401,247 pre-computed positions that were known to be either wins, losses, or draws (Lake, Schaeffer, and Lu, 1994). These data were available at runtime during the look-ahead search, which allowed CHINOOK to enter into lines of play that would avoid losses (within its horizon of search) as well discover deep, subtle wins.

Having such game-theoretical values (GTV) available for the search engine at runtime is extremely valuable, but in certain cases it is not enough information to procure the win. Section 3.2 showcases some 7-piece positions that are wins for the side to move but cannot be won using only a database with the game-theoretical values stored. While such a database recognizes the wins as it builds its tree during the search, it cannot determine the winning sequence. A database with information associated with the distance until a conversion (capture of a piece, or promotion of a checker to a king) takes place is of some help. Examples are presented that demonstrate the power of such a Distance To Conversion database, as well as some of the weaknesses.

This paper is organized as follows. Section 2 presents an overview of the three different types of checkers endgame databases, and briefly tabulates the pros and cons of each category. Section 3 contains the solution to the longest 7-piece database win and a comprehensive listing of all of the data collected in the 2-to-7-piece Perfect Play Lookup (PPL) databases. The longest wins are presented in each sub-database grouping. Section 4 demonstrates several improvements to a very common checkers endgame known as Fourth Position. This ending was first published in 1756 and has been studied by the tournament checkers playing community ever since. It should be noted that the PPL database solution begins in such an unorthodox fashion that it is worthy of special attention. Section 5 offers a brief conclusion regarding what has been learned, particularly about the complexity of the game of checkers.
By improving upon the play of a common checkers endgame first published 247 years ago that has since been studied and analyzed by the strongest human players in the history of checkers, this paper asserts that the PPL database is capable of outperforming the world’s best human players from any time period.

2. Overview of Different Types of Databases

There are three different ways to catalog checkers information that can be useful to a program. Each type of database has benefits and drawbacks, which are summarized in Table 1.

<table>
<thead>
<tr>
<th>Database</th>
<th>Benefits</th>
<th>Drawbacks</th>
</tr>
</thead>
</table>
| Game Theoretical Value (GTV) | 1) Easiest type of database to compute.  
2) Can be generated quickly.  
3) A post-process routine can compress the data efficiently allowing for runtime probing to assist the Alpha-Beta search engine. | 1) Once reaching a database position over the board, no information is available regarding the best way to proceed.  
2) Positions that are theoretically won can in practice be drawn by repetition since the winning path cannot be found. |
| Distance To Conversion (DTC) | 1) Can be computed about as easily as a GTV database.  
2) In King-heavy endings the play can mirror PPL database results.  
3) A win can always be achieved, even if it takes much longer than a PPL database’s solution.  
4) Positions with draws as the result can be removed from the database. | 1) Requires much more RAM and disk space to compute compared to a GTV database.  
2) The database prefers a known conversion path which may take longer to win than a potentially much shorter path to victory known by a PPL database.  
3) With fewer kings on the board, playing precision is much lower than that of a PPL database.  
4) Post-process compression is not nearly as good as that of a GTV database. |
| Perfect Play Lookup (PPL) | 1) Always wins by selecting the shortest possible route.  
2) Always capable of postponing losses for as long as possible.  
3) Positions with draws as the result can be removed from the database.  
4) The verification routine virtually guarantees that the GTV database used in the process is correct. | 1) Difficult to compute both in algorithm complexity and time requirements.  
2) Requires much more RAM and disk space to compute compared to a GTV database.  
3) Post-process compression is not nearly as good as that of a GTV database (but can be as good as that of a DTC database). |

Table 1. Benefits and drawbacks of the different types of checkers databases.
2.1 Application of a GTV Database

Typically, a large database of game-theoretical-value information can be probed at run time, greatly reducing the number of nodes that need to be evaluated by a search engine. Once a database position is encountered in RAM, no additional move generation or merit assignments need to be invoked. The GTV score is returned, and that particular leaf node has perfect information attached to it. Schaeffer, Lake, Lu, and Bryant (1996) have demonstrated the benefits of this approach with their program CHINOOK, which won the World “Man versus Machine” Championship in 1994 and successfully defended its title in 1995. In probing such a database, the search tree can return valuable information while still a great distance away. The examples in Figures 1 to 3 demonstrate how a 12-piece position can be evaluated as a forced win with only a six-piece database accessible in RAM.

![Figure 1](image)

**Figure 1.** Black to move wins by forcing a trade into a won six-piece database position.

As other jump paths are examined, the position shown in Figure 3 will be reached after 14-17 21x14, 20-24 28x19, 6-9 13x6, 1x10x17x26. The 6-piece database position shown in Figure 3 is a loss for White to move, so the score for Black to move from the parent position is returned as a win. Since all captures are forced in the game of checkers, the program will elect to enter into the inescapable line of play leading to Figure 3. In so doing, the program will properly announce a win from the 12-piece position in Figure 1, and play the move 14-17.

A quick glance at Figures 2 and 3 will not reveal anything obvious to the casual player, but after conducting a search the correct result becomes evident. The most outstanding feature of the GTV information is that no such search ever needs to be performed once a position in the database is
found. This is the equivalent of “extending the search” from the database position to the terminal node many plies distant.

![Figure 2. White to move draws.](image)

![Figure 3. White to move loses.](image)

In the case of Figure 3, the PPL database indicates that White loses after a total of 102 plies. So, a GTV database, by identifying that position as a theoretical win, performed the functional equivalent of searching 102 plies and returning a score indicating a loss would occur. Notice that the number of pieces in a GTV database can be relatively small yet it can still effectively direct the search from a position with many more pieces. However, as will be shown in Subsection 3.2, once a program with only a GTV database is actually in a won position, in certain cases it can be sufficiently difficult to converge on the win.

### 2.2 Creating a DTC Database

A DTC database will store the number of plies for each position until either a checker crowns or a piece is captured. It does not have any information to distinguish which result is achieved as the goal; it only knows it is heading for one or the other. Unlike a GTV database, which can represent four positions in each byte during the computation (and five positions per byte after the computation prior to compression) the DTC database needs one entire byte for each position in order to store a conversion range from 0 to 255 plies.

The process of creating the DTC database is nearly identical to the GTV database. If the DTC database is also being used as a GTV database, then two of the eight bits in the byte must be reserved for the win-loss-draw assessment, leaving only six bits (0-63) available for the maximum depth to conversion. There is a way to double this ply count if you divide the actual
depth by two, and take note that if the side to move wins, the plycount must 
be an odd number, since that side makes the last move. If the side to move 
loses, it makes the second to last move, which always must be an even 
number. Therefore, to “decompress” the true ply count, double the number 
that is stored, then add one to it if the position is a win. If it is a loss, there is 
no need to change the number. If it is a draw, the depth until conversion is 
meaningless. If the DTC database is created as a separate post-process, and 
the GTV database is used to determine the win-loss-draw status of a 
position, then all eight bits can be used to store conversion information. 
Using the aforementioned division-by-two schema a maximum conversion 
depth of 511 plies can be stored.

The iteration process for the DTC database begins with the jump pass, 
which is the same as would be performed with a GTV database, with one 
notable difference. The DTC database stores the result for each jumping win 
as a “1-ply” conversion. Therefore the counter for each win in which a jump 
exists will be set to 1. Next, any crowning moves are generated, and 
independent of the win or loss result, these are stored as a conversion in 1 
ply as well. Thereafter, as each pass over the database is made, whenever a 
win or loss result is able to be determined, the iteration number is stored in 
the database. The idea is that more difficult positions (presumably) will have 
conversion iteration counts greater than positions that are near the 
conversion horizon.

When the computation is completed, one goes on to the next slice, but the 
conversion information for the crowning moves or jumps is not inherited. 
Each database slice is computed independently of all others. No conversion 
information is shared across databases.

2.3 Weaknesses of DTC Databases

In difficult positions where there is a majority of Kings, a DTC database 
is most valuable. As more Checkers are introduced into a position, the 
probability that a DTC database will make the same, highly accurate move 
as the PPL database diminishes. Even in elementary positions like the one 
shown below in Figure 4, a DTC database will not play a move that is 
obvious to any ordinary player.

Even novice checker players will make the move 18-23 in Figure 4, 
winning after White makes any move with the King, but a DTC database 
will not.
The glaring weakness of a DTC database centers on a potential “one ply conversion horizon” that can arise during a computation cycle. A move that converts in fewer plies but takes much longer to win could be preferred over a shorter win that takes longer to convert. With a quick glance anyone can see that in Figure 4 the sequence 18-23 31-26 23x30 wins, as does 18-23 31-27 23x32. This requires three plies of information, which the PPL database does have, but which the DTC database does not have. The DTC database will “know” that 25-29 converts in 1 ply, and 25-30 converts in 1 ply. When the move 18-23 is examined, it will not lead to an immediate conversion, and therefore will be deemed to be inferior. The DTC database will have stored a value indicating that the position after 18-23 results in a conversion in 2 more plies. This conversion into the “2 against 0” database will win, of course, but the DTC has no information regarding the “distance to win”.

2.4 Creating a PPL Database

Unlike a DTC database, a PPL database stores complete information about the line of play leading all the way to the terminal (lost) position. It does this by backing up and storing the number of plies to win or lose for every won or lost position during the database generation process, starting with the 2-piece database. Like the DTC database, an entire byte is required for each position in order to store this move-to-win (MTW) information.

Computation of a PPL database is much more difficult, both algorithmically and in terms of the amount of calculation required, than that of either a GTV or DTC database. This is due to the fact that one is not necessarily done once an MTW value has been assigned to a position. During the computation of a GTV or DTC database, once a GTV (win/loss/draw) or DTC (iteration count) value has been assigned to a position, no further computation is required. During the computation of a PPL database, the MTW values are subject to change from one iteration to the next. The MTV values are backed up through a tree of possible lines of play, and this tree dynamically changes as a function of the iteration depth. This process essentially amounts to a complex sorting procedure which
cannot be terminated until a pass is made over the entire database that produces no changes in any of the MTW values.

The sorting procedure works as follows. In a won position all moves are generated, and the resulting positions that lose for the other side are queried for their MTW values (which may not yet exist or be reliable, depending on the position and the current iteration). The smallest value is then backed up into the parent position. This represents the move that will result in the quickest win. In a lost position all moves are generated, and the largest resulting MTW value is backed up into the parent position. (Recall that in a lost position all moves must lead to wins for the other side.) This represents the move that will result in the longest, most drawn-out loss. The database generation program makes repeated passes over the database slice being computed, and the sorting process continues until none of the MTW values changes.

It should be noted that there are some lines of play that lead to wins with no “conversion” taking place, and this must be taken into account during the generation of the database. This situation occurs when a move is made that blocks the opponent so he cannot move. A loss in the game of checkers occurs when one side cannot move, due to not having any pieces remaining, or having pieces on the board that are blocked in such a way that no moves are available. The blocking move leading to the win may or may not involve a capture or a promotion (i.e., a conversion).

3. The Longest 7-Piece Database Win

Figure 5 contains the longest 7-piece database win. Below we provide the PPL database solution.

Listing 1. The PPL database solution to the longest 7-piece win.

### 3.1 Perfect Play Data

Table 2 lists statistics on all of the 2-to-7 piece database slices that were solved with four or fewer pieces for one side and Black to move. In it is shown the total number of positions as a function of the database slice, the number of plies associated with the longest win and loss for each slice, and one position for a longest win for Black to move. In some cases, as the material distribution becomes more dominant for one side, the longest win features a position with a forced jump for the weaker side to move. After a bit of reflection, this result makes sense. With the strong side to move, the win will precipitate very quickly. The weak side to move will execute a jump, perhaps equalizing or even surpassing the material of the former strong side of the board, then the total number of plies to win from the resulting sub-database will substantially add to the length of the game. In the “Position” column, BK = Black King, WK = White King, BC = Black Checker, WC = White Checker.

<table>
<thead>
<tr>
<th>Material Distribution</th>
<th>Total Positions</th>
<th>Longest Win/Loss</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K + 0C vs. 1K + 0C</td>
<td>992</td>
<td>11/10</td>
<td>BK: 4; WK: 29</td>
</tr>
<tr>
<td>1K + 0C vs. 0K + 1C</td>
<td>868</td>
<td>11/10</td>
<td>BK: 32; WC: 20</td>
</tr>
<tr>
<td>0K + 1C vs. 1K + 0C</td>
<td>868</td>
<td>5/12</td>
<td>BC: 14; WK: 26</td>
</tr>
<tr>
<td>0K + 1C vs. 0K + 1C</td>
<td>760</td>
<td>13/12</td>
<td>BC: 25; WC: 30</td>
</tr>
<tr>
<td>2K + 0C vs. 1K + 0C</td>
<td>14,880</td>
<td>33/34</td>
<td>BK: 1,2; WK: 19</td>
</tr>
<tr>
<td>2K + 0C vs. 0K + 1C</td>
<td>13,020</td>
<td>33/34</td>
<td>BK: 1,2; WC: 19</td>
</tr>
<tr>
<td>1K + 1C vs. 1K + 0C</td>
<td>26,040</td>
<td>47/48</td>
<td>BK: 32; BC: 4; WK: 23</td>
</tr>
<tr>
<td>1K + 1C vs. 0K + 1C</td>
<td>22,800</td>
<td>47/48</td>
<td>BK: 32; BC: 4; WC: 15</td>
</tr>
<tr>
<td>0K + 2C vs. 1K + 0C</td>
<td>11,340</td>
<td>61/62</td>
<td>BC: 3,4; WK: 26</td>
</tr>
<tr>
<td>0K + 2C vs. 0K + 1C</td>
<td>9,936</td>
<td>61/62</td>
<td>BC: 3,4; WC: 26</td>
</tr>
<tr>
<td>2K + 0C vs. 2K + 0C</td>
<td>215,760</td>
<td>49/48</td>
<td>BK: 26,30; WC: 29,31</td>
</tr>
<tr>
<td>2K + 0C vs. 1K + 1C</td>
<td>377,580</td>
<td>95/94</td>
<td>BK: 2,3; WK: 21; WC: 25</td>
</tr>
<tr>
<td>2K + 0C vs. 0K + 2C</td>
<td>164,430</td>
<td>89/92</td>
<td>BK: 28,31; WC: 6,30</td>
</tr>
<tr>
<td>1K + 1C vs. 1K + 1C</td>
<td>661,200</td>
<td>103/102</td>
<td>BK: 28; BC: 18; WK: 3; WC: 29</td>
</tr>
<tr>
<td>1K + 1C vs. 0K + 2C</td>
<td>288,144</td>
<td>107/108</td>
<td>BK: 28; BC: 4; WC: 27,30</td>
</tr>
<tr>
<td>0K + 2C vs. 0K + 2C</td>
<td>125,664</td>
<td>109/108</td>
<td>BC: 4,24; WC: 29,30</td>
</tr>
<tr>
<td>3K + 0C vs. 1K + 0C</td>
<td>143,840</td>
<td>29/30</td>
<td>BK: 7,16,29; WK: 11</td>
</tr>
<tr>
<td>3K + 0C vs. 0K + 1C</td>
<td>125,860</td>
<td>27/28</td>
<td>BK: 28,31,32; WC: 19</td>
</tr>
<tr>
<td>2K + 1C vs. 1K + 0C</td>
<td>377,580</td>
<td>41/38</td>
<td>BK: 7,16; BC: 8; WK: 3</td>
</tr>
<tr>
<td>2K + 1C vs. 0K + 1C</td>
<td>330,600</td>
<td>37/32</td>
<td>BK: 29,30; BC: 25; WC: 31</td>
</tr>
<tr>
<td>1K + 2C vs. 1K + 0C</td>
<td>328,860</td>
<td>53/54</td>
<td>BK: 19; BC: 9,10; WK: 15</td>
</tr>
<tr>
<td>1K + 2C vs. 0K + 1C</td>
<td>288,144</td>
<td>41/42</td>
<td>BK: 32; BC: 9,14; WC: 13</td>
</tr>
<tr>
<td>0K + 3C vs. 1K + 0C</td>
<td>95,004</td>
<td>59/58</td>
<td>BC: 4,7,8; WK: 3</td>
</tr>
<tr>
<td>0K + 3C vs. 0K + 1C</td>
<td>83,304</td>
<td>55/56</td>
<td>BC: 7,8,11; WC: 12</td>
</tr>
<tr>
<td>3K + 0C vs. 2K + 0C</td>
<td>2,013,760</td>
<td>67/68</td>
<td>BK: 8,29,30; WK: 12,18</td>
</tr>
<tr>
<td>3K + 0C vs. 1K + 1C</td>
<td>3,524,080</td>
<td>89/90</td>
<td>BK: 24,16,12; WK: 20; WC: 29</td>
</tr>
<tr>
<td>3K + 0C vs. 0K + 2C</td>
<td>1,530,680</td>
<td>93/88</td>
<td>BK: 25,26,29; WC: 9,30</td>
</tr>
<tr>
<td>1K + 2C vs. 2K + 0C</td>
<td>5,286,120</td>
<td>147/148</td>
<td>BK: 4; BC: 5; WK: 26,30</td>
</tr>
<tr>
<td>2K + 1C vs. 1K + 1C</td>
<td>9,256,800</td>
<td>139/140</td>
<td>BK: 4,30; BC: 5; WK: 22; WC: 10</td>
</tr>
<tr>
<td>2K + 1C vs. 0K + 2C</td>
<td>4,034,016</td>
<td>111/140</td>
<td>BK: 28; BC: 4,8; WC: 7,12</td>
</tr>
<tr>
<td>1K + 2C vs. 0K + 2C</td>
<td>1,330,056</td>
<td>155/154</td>
<td>BC: 1,3,4; WK: 5,26</td>
</tr>
<tr>
<td>0K + 3C vs. 1K + 1C</td>
<td>2,332,512</td>
<td>161/162</td>
<td>BC: 1,4,5; WK: 14; WC: 24</td>
</tr>
<tr>
<td>0K + 3C vs. 0K + 2C</td>
<td>1,018,056</td>
<td>155/160</td>
<td>BC: 5,7,9; WC: 6,26</td>
</tr>
<tr>
<td>4K + 0C vs. 1K + 0C</td>
<td>1,006,880</td>
<td>29/30</td>
<td>BK: 9,17,26,27; WK: 22</td>
</tr>
<tr>
<td>4K + 0C vs. 0K + 1C</td>
<td>881,020</td>
<td>23/24</td>
<td>BK: 4,28,29,32; WC: 23</td>
</tr>
<tr>
<td>3K + 1C vs. 1K + 0C</td>
<td>3,524,080</td>
<td>29/30</td>
<td>BK: 17,26,27; BC: 9; WK: 22</td>
</tr>
<tr>
<td>3K + 1C vs. 0K + 1C</td>
<td>3,085,600</td>
<td>25/26</td>
<td>BK: 28,31,32; BC: 24; WC: 19</td>
</tr>
<tr>
<td>2K + 2C vs. 1K + 0C</td>
<td>4,604,040</td>
<td>37/38</td>
<td>BK: 31,32; BC: 27,28; WK: 24</td>
</tr>
<tr>
<td>2K + 2C vs. 0K + 1C</td>
<td>4,034,016</td>
<td>31/28</td>
<td>BK: 31,32; BC: 27,28; WC: 30</td>
</tr>
<tr>
<td>1K + 3C vs. 1K + 0C</td>
<td>2,660,112</td>
<td>43/44</td>
<td>BK: 26; BC: 4,11,19; WK: 23</td>
</tr>
<tr>
<td>1K + 3C vs. 0K + 1C</td>
<td>2,332,512</td>
<td>39/40</td>
<td>BK: 4; BC: 7,8,11; WC: 12</td>
</tr>
<tr>
<td>0K + 4C vs. 1K + 0C</td>
<td>573,300</td>
<td>51/52</td>
<td>BC: 7,8,11,15; WK: 12</td>
</tr>
<tr>
<td>0K + 4C vs. 0K + 1C</td>
<td>503,100</td>
<td>49/50</td>
<td>BC: 4,7,8,11; WC: 12</td>
</tr>
<tr>
<td>3K + 0C vs. 3K + 0C</td>
<td>18,123,840</td>
<td>73/74</td>
<td>BK: 3,12,23; WK: 16,31,32</td>
</tr>
<tr>
<td>3K + 0C vs. 2K + 1C</td>
<td>47,575,080</td>
<td>147/146</td>
<td>BK: 3,8,15; WC: 7,22; WC: 28</td>
</tr>
<tr>
<td>3K + 0C vs. 1K + 2C</td>
<td>41,436,360</td>
<td>151/150</td>
<td>BK: 1,8,15; WK: 7; WC: 28,29</td>
</tr>
<tr>
<td>3K + 0C vs. 0K + 3C</td>
<td>11,970,504</td>
<td>149/150</td>
<td>BK: 13,26,28; WC: 5,14,29</td>
</tr>
<tr>
<td>2K + 1C vs. 3K + 0C</td>
<td>47,575,080</td>
<td>147/146</td>
<td>BK: 11,26; BC: 5; WK: 18,25,30</td>
</tr>
<tr>
<td>2K + 1C vs. 2K + 1C</td>
<td>124,966,800</td>
<td>153/152</td>
<td>BK: 10,19; BC: 1; WK: 2,6; WC: 31</td>
</tr>
<tr>
<td>2K + 1C vs. 1K + 2C</td>
<td>108,918,432</td>
<td>161/162</td>
<td>BK: 25,18; BC: 1; WK: 17; WC: 24,28</td>
</tr>
<tr>
<td>2K + 1C vs. 0K + 3C</td>
<td>31,488,912</td>
<td>155/160</td>
<td>BK: 15,27; BC: 22; WC: 23,28,32</td>
</tr>
<tr>
<td>1K + 2C vs. 3K + 0C</td>
<td>41,436,360</td>
<td>151/150</td>
<td>BK: 26; BC: 4,5; WK: 18,25,32</td>
</tr>
<tr>
<td>1K + 2C vs. 2K + 1C</td>
<td>108,918,432</td>
<td>161/162</td>
<td>BK: 16; BC: 5,9; WK: 8,15; WC: 32</td>
</tr>
<tr>
<td>1K + 2C vs. 1K + 2C</td>
<td>95,001,984</td>
<td>167/166</td>
<td>BK: 25; BC: 5,9; WK: 18; WC: 17,30</td>
</tr>
<tr>
<td>1K + 2C vs. 0K + 3C</td>
<td>27,487,512</td>
<td>163/164</td>
<td>BK: 14; BC: 5,6; WC: 25,19,12</td>
</tr>
<tr>
<td>0K + 3C vs. 3K + 0C</td>
<td>11,970,504</td>
<td>149/150</td>
<td>BK: 4,19,28; WK: 5,7,20</td>
</tr>
<tr>
<td>0K + 3C vs. 2K + 1C</td>
<td>31,488,912</td>
<td>155/160</td>
<td>BC: 1,5,10; WC: 6,18; WC: 11</td>
</tr>
<tr>
<td>0K + 3C vs. 1K + 2C</td>
<td>27,487,512</td>
<td>163/164</td>
<td>BC: 8,14,21; WK: 19; WC: 27,28</td>
</tr>
<tr>
<td>0K + 3C vs. 0K + 3C</td>
<td>79,59,904</td>
<td>161/162</td>
<td>BC: 1,2,3; WC: 14,17,19</td>
</tr>
<tr>
<td>Move Description</td>
<td>Solutions</td>
<td>Time (BK, WK)</td>
<td>GTV Positions</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>4K + 0C vs. 2K + 0C</td>
<td>13,592,880</td>
<td>67/68</td>
<td>BK: 4,12,29,30; WK: 11,22</td>
</tr>
<tr>
<td>4K + 0C vs. 1K + 1C</td>
<td>23,787,540</td>
<td>87/88</td>
<td>BK: 9,10,19,27; WK: 15; WC: 30</td>
</tr>
<tr>
<td>4K + 0C vs. 0K + 2C</td>
<td>10,359,090</td>
<td>51/44</td>
<td>BK: 17,19,26,27; WC: 20,31</td>
</tr>
<tr>
<td>3K + 1C vs. 2K + 0C</td>
<td>47,575,080</td>
<td>135/114</td>
<td>BK: 9,17,29; BC: 5; WK: 13,17</td>
</tr>
<tr>
<td>3K + 1C vs. 1K + 1C</td>
<td>83,311,200</td>
<td>91/88</td>
<td>BK: 17,25,26; BC: 19; WK: 20; WC: 29</td>
</tr>
<tr>
<td>3K + 1C vs. 0K + 2C</td>
<td>36,306,144</td>
<td>95/66</td>
<td>BK: 19,25,27; BC: 17; WC: 30,31</td>
</tr>
<tr>
<td>2K + 2C vs. 2K + 0C</td>
<td>62,154,540</td>
<td>147/138</td>
<td>BK: 14,29; BC: 5,26; WK: 30,31</td>
</tr>
<tr>
<td>2K + 2C vs. 1K + 1C</td>
<td>108,918,432</td>
<td>143/140</td>
<td>BK: 26,27; BC: 5,13; WK: 14; WC: 31</td>
</tr>
<tr>
<td>2K + 2C vs. 0K + 2C</td>
<td>47,500,992</td>
<td>99/80</td>
<td>BK: 17,19; BC: 4,26; WC: 30,31</td>
</tr>
<tr>
<td>1K + 3C vs. 2K + 0C</td>
<td>35,911,512</td>
<td>149/150</td>
<td>BK: 26; BC: 5,6,7; WK: 1,31</td>
</tr>
<tr>
<td>1K + 3C vs. 1K + 1C</td>
<td>62,977,824</td>
<td>153/154</td>
<td>BK: 9; BC: 4,5,8; WK: 10; WC: 13</td>
</tr>
<tr>
<td>1K + 3C vs. 0K + 2C</td>
<td>27,487,512</td>
<td>109/146</td>
<td>BK: 28; BC: 4,8,11; WC: 7,12</td>
</tr>
<tr>
<td>0K + 4C vs. 2K + 0C</td>
<td>7,739,550</td>
<td>155/156</td>
<td>BC: 1,4,8,18; WK: 10,26</td>
</tr>
<tr>
<td>0K + 4C vs. 1K + 1C</td>
<td>13,583,700</td>
<td>153/154</td>
<td>BC: 1,6,16,19; WK: 14; WC: 20</td>
</tr>
<tr>
<td>0K + 4C vs. 0K + 2C</td>
<td>5,933,850</td>
<td>153/148</td>
<td>BC: 5,7,10,14; WC: 15,19</td>
</tr>
<tr>
<td>0K + 2C vs. 0K + 4C</td>
<td>5,933,850</td>
<td>153/148</td>
<td>BC: 5,7,10,14; WC: 15,19</td>
</tr>
<tr>
<td>4K + 0C vs. 3K + 0C</td>
<td>117,804,960</td>
<td>113/114</td>
<td>BK: 3,4,5,26; WK: 1,11,15</td>
</tr>
<tr>
<td>4K + 0C vs. 2K + 1C</td>
<td>309,238,020</td>
<td>149/142</td>
<td>BK: 2,29; BC: 5; WK: 6,7,8,30</td>
</tr>
<tr>
<td>4K + 0C vs. 1K + 2C</td>
<td>269,336,340</td>
<td>149/148</td>
<td>BK: 3; BC: 5,15; WK: 6,18,25,30</td>
</tr>
<tr>
<td>4K + 0C vs. 0K + 3C</td>
<td>77,808,276</td>
<td>151/150</td>
<td>BC: 5,9,11; WK: 16,24,28,29</td>
</tr>
<tr>
<td>3K + 1C vs. 3K + 0C</td>
<td>412,317,360</td>
<td>207/208</td>
<td>BK: 21,28,30; BC: 3; WK: 1,22,32</td>
</tr>
<tr>
<td>3K + 1C vs. 2K + 1C</td>
<td>1,083,045,600</td>
<td>201/202</td>
<td>BK: 1,29,30; BC: 24; WK: 31,27; WC: 11</td>
</tr>
<tr>
<td>3K + 1C vs. 1K + 2C</td>
<td>943,959,744</td>
<td>153/144</td>
<td>BK: 10; BC: 4,5; WK: 14,18,25; WC: 9</td>
</tr>
<tr>
<td>3K + 1C vs. 0K + 3C</td>
<td>272,903,904</td>
<td>159/158</td>
<td>BC: 3,5,9; WK: 8,16,18; WC: 31</td>
</tr>
<tr>
<td>2K + 2C vs. 3K + 0C</td>
<td>538,672,680</td>
<td>245/246</td>
<td>BK: 4,11; BC: 2,5; WK: 3,10,29</td>
</tr>
<tr>
<td>2K + 2C vs. 2K + 1C</td>
<td>1,415,939,616</td>
<td>241/240</td>
<td>BK: 4,32; BC: 5,8; WK: 17,23; WC: 12</td>
</tr>
<tr>
<td>2K + 2C vs. 1K + 2C</td>
<td>1,235,025,792</td>
<td>191/192</td>
<td>BK: 5,27; BC: 12,20; WK: 19; WC: 11,32</td>
</tr>
<tr>
<td>2K + 2C vs. 0K + 3C</td>
<td>357,337,656</td>
<td>161/166</td>
<td>BC: 2,5,12; WK: 3,16; WC: 9,31</td>
</tr>
<tr>
<td>1K + 3C vs. 3K + 0C</td>
<td>311,233,104</td>
<td>249/248</td>
<td>BK: 6; BC: 1,18,15; WK: 5,14,16</td>
</tr>
<tr>
<td>1K + 3C vs. 2K + 1C</td>
<td>818,711,712</td>
<td>253/252</td>
<td>BK: 4; BC: 1,8,10; WK: 9,21; WC: 12</td>
</tr>
<tr>
<td>1K + 3C vs. 1K + 2C</td>
<td>714,675,312</td>
<td>237/238</td>
<td>BK: 5; BC: 7,8,9; WK: 27; WC: 6,19</td>
</tr>
<tr>
<td>1K + 3C vs. 0K + 3C</td>
<td>206,957,504</td>
<td>183/198</td>
<td>BK: 29; BC: 5,7,8; WC: 12,24,30</td>
</tr>
<tr>
<td>0K + 4C vs. 3K + 0C</td>
<td>67,076,100</td>
<td>233/230</td>
<td>BC: 2,4,15,27; WK: 5,16,32</td>
</tr>
<tr>
<td>0K + 4C vs. 2K + 1C</td>
<td>176,588,100</td>
<td>249/248</td>
<td>BC: 1,2,4,6; WK: 28,32; WC: 27</td>
</tr>
<tr>
<td>0K + 4C vs. 1K + 2C</td>
<td>154,280,100</td>
<td>243/242</td>
<td>BC: 4,5,6,8; WK: 28; WC: 12,27</td>
</tr>
<tr>
<td>0K + 4C vs. 0K + 3C</td>
<td>44,717,500</td>
<td>209/210</td>
<td>BC: 4,5,7,11; WC: 6,19,32</td>
</tr>
</tbody>
</table>

Table 2. Positions with the longest solutions for the 2- to-7-piece databases.

### 3.2 Difficult Theoretical Wins

It is possible to be in a position that is a theoretical win that is too difficult for a program to win even while consulting a GTV database (see Figures 6 and 7). This is due to several factors; we mention three of them.

1. The program will never make a move that loses or gives away the draw on any given turn as it consults the GTV databases, but sometimes every
move in a position will win. The GTV database is of no help in reducing the size of the game tree in these instances.

2. The complete path to the win might not require the use of any of the intermediate goals in the evaluation function of the checkers program, so as long as the solution is beyond the horizon of the search, the win could be postponed.

3. King-heavy positions will produce principal variations in which the Kings tend to wander and gain nothing if forced trades can be avoided.

Figure 6. A longest conversion win with 3 Kings and 1 Checker versus 3 Kings. Black to move can force a trade after 149 plies, starting with the move 27-24.

Figure 7. A longest win with 3 Kings and 1 Checker versus 3 Kings. Moving the Checker on square 3 will result in a Draw. Only 28-24 will win.

Figure 6 comes from the DTC database of Murray Cash (Cash and Miller, 2002) and is listed as the longest conversion win in the 7-piece database in which a Checker remains unmoved. Comparing this position to Figure 19, page 248 from Schaeffer (1997) we note that Schaeffer had two of the white Kings on squares 12 and 15 instead of 16 and 19. The Schaeffer position and the Cash position both require 207 plies to win.

Figure 7 is from the Dodgen-Trice PPL database, showing another "longest win" possible in the same database slice as the position in Figure 6. Listings 2 and 3 show how the PPL database will play each position.

Listing 2. The PPL database solution to Figure 6.

The 7-Piece Perfect Play Lookup Database for the Game of Checkers

Listing 3. The PPL database solution to Figure 7.

Listing 4. The PPL database defends the weak side of Figure 6 against a GTV database on the winning side, and a draw ensues via repetition.

Listing 5. The PPL database defends the weak side of Figure 7 against a GTV database on the winning side, and a different kind of draw is reached. The positions will repeat in a cycle every 66 plies.
It is interesting to note that after move 12 in Listing 5, which is from Figure 7, the same type of position as Figure 6 is created; i.e., one in which the Checker cannot crown since it is being blocked by an enemy King. Even with this common theme, two different types of draws result. In Listing 4, a “see-saw” draw occurs when the hash table saturates and moves leading to the win have all been played before. Recall one of the uses of the hash table is to score repeated moves of the same King as a draw, so that you do not shuffle the same piece back and forth twenty times and believe you have conducted a valid 40 ply search (20 for one side, 20 for the other). Likewise, arriving at the same position many times during the search via transposition without making progress should be discouraged. The program ends up in the undesirable situation where most or all of the winning lines are found in the hash table, yet they are the only subset of moves that will win. In Listing 5, a lengthy cycle from moves 55 to 88 could theoretically repeat ad infinitum, starting at move 89. Without being able to search at least 67 plies into the future from move 55, this cycle cannot be avoided.

### 3.3 GTV Database Program vs. PPL Database Program

An experiment was performed to observe how two different Grandmaster-level programs (Waldteufel, 2002; Gilbert, 2002) would play against the PPL database from a “longest win” test position. In each case, the WORLD CHAMPIONSHIP CHECKERS (WCC) program (Dodgen and Trice, 2001) with the PPL database played the losing side of each ending, and both the WYLLIE program and KINGSROW program played the winning side. All of the programs had access to a GTV database probed in RAM during the search that contained at least all of the 19,055,258,760 7-piece positions featuring four against three. The WCC program consulted the PPL database when defending the weak side on every move. The starting position for each
game was the position shown in Figure 5. In this position, Black to move can win in 253 plies.

**Listing 6.** WYLLIE program vs. WCC, February 7, 2003. WCC was able to draw from the losing starting position shown in Figure 5.

<table>
<thead>
<tr>
<th>Moves</th>
<th>Result</th>
</tr>
</thead>
</table>

**Listing 7.** KINGSROW program vs. WCC, March 3, 2003. WCC was able to draw from the losing starting position shown in Figure 5.

<table>
<thead>
<tr>
<th>Moves</th>
<th>Result</th>
</tr>
</thead>
</table>

The WYLLIE program searched for 10 to 15 seconds per move, averaging about 620,000 moves per second during the search. The time duration was chosen for practical purposes. This ending was very long, and if we took a combined 30 seconds to type our moves to one another, the game would last over an hour if 250 plies were required to win.

The WYLLIE program was unable to win this ending, conceding the draw after 196 plies of play. Even after this lengthy engagement, the PPL database indicated that the win was 177 plies away for the WYLLIE program. In this respect, only 96 plies of progress were observed after 196 plies of actual play. The WYLLIE program got as close as 135 plies from the terminal
position before it began to make non-optimal moves. Listing 6 shows the moves made by WYLLIE and WCC during this game.

The KINGSROW program also played with an average search time of 10-15 seconds per move, which was increased to 30 seconds per move (upon request of the KINGSROW programmer) once three Kings were on the board for the winning side. The program was able to get an average search depth of 33 plies during the course of play. The KINGSROW program got as close as 159 plies from the terminal position, but it too started to make non-optimal moves allowing WCC to push the win further away. At the end of 164 plies of play, the win was still 181 plies distant, so KINGSROW netted 72 plies of progress after 164 plies. Listing 7 shows the moves made by KINGSROW and WCC during this game.

4. Improving Play from the Fourth Position Ending

There is an arrangement on the checkerboard in which the weak side can have one less King than the strong side and still retain a draw. This study problem was designated Fourth Position by the checkers fraternity. With one slight modification to the arrangement of this position, or by altering the side to move, the strong side gains the ability to procure a win which requires precise timing of the disposal of one of its pieces.

Winning the textbook form of Fourth Position requires 81 plies according to the PPL database. The original published play from 1756 features a first move that would require a total of 85 plies to complete the win, but there was also a sub-optimal defensive move in this analysis. The sub-optimal line would surrender the game 12 plies more quickly than would the PPL database.

The original solution entailed the well-motivated retreat 22-18 (Payne, 1756) to start things off, but the PPL database offers 22-25! winning more quickly. Even more incredible is the fact that two moves later Black will play 25-29, a move that is usually strongly discouraged in almost every position in which White has a Checker on square 30. The improved PPL solution is presented in Listing 8. A subset of the classic solution to Fourth Position is presented thereafter, with commentary correcting the play on the defensive side.
The 7-Piece Perfect Play Lookup Database for the Game of Checkers

Listing 8. The PPL solution to Fourth Position from Figure 8.


Figure 9. White to move after 22-18, 31-27, 23-19 from Figure 8.

The PPL database was able to identify some play on the weak side of Fourth Position that was not optimal. Figure 9 shows the position with White to move after: 22-18, 31-27, 23-19 which has traditionally been followed by the retreat of the King 27-31. The perfect play database announces that this move leads to a win in 69 plies for Black, but the optimal line will persist for 12 plies longer. The best defense from the position shown in Figure 9 is 32-28, 18-22 (heading back to square 29, as suggested by the original improvement, is still the fastest course of action from here) 27-31, 22-25, 31-27, 25-29, 27-31, 20-24, 28-32, 24-28, 31-27, 29-25, 27-24, 19-16, 24-27, 16-20, 27-23, 25-22, 23-27, 22-26, 30x23, 28-24, 32-28 and now 24x31 leads a 5-piece position that White will lose in 58 plies.

The purpose of the move 32-28 is to prevent the immediate 19-24 by Black. It should be noted that as long as White keeps the King on square 28, Black cannot play the strong 19-24 attack, which is instrumental in concluding the game more quickly. It is not the absence of 27-31 on move two for White that extends the life of the weak side, it is the presence of the move 32-28.

5. Conclusions

The game of checkers is deceptive in its apparent simplicity. Most strong contemporary checkers programs have large opening books capable of circumventing early losses, and are likewise capable of handling the tactics in the middle game beyond the ability of the strongest human players. But, as was demonstrated, the endgame domain is still sufficiently complex so as to prevent grandmaster-level programs from winning in positions that are known wins with as few as seven pieces on the board. This result was rather
surprising and should underscore the complexity inherent in the game of checkers.

The Perfect Play databases of Dodgen and Trice are the only databases in existence that allow a software program to play the game of checkers perfectly in the endgame. The 7-piece perfect play lookup database allows the WORLD CHAMPIONSHIP CHECKERS program to announce a win from a distance of 253 plies.

We will continue to build larger PPL databases as time and personal computer resources will allow. The web site at WorldChampionshipCheckers.com will showcase the PPL database building progress and other items of interest to checkers and programming enthusiasts.

Acknowledgements

Without the pioneering effort of the CHINOOK programming team, and in particular Dr. Jonathan Schaeffer and Robert Lake, our work in the domain of endgame databases for the game of checkers would never have gotten off the ground. The concept of applying an indexing function to create a sparsely populated matrix for the game of checkers is one of the critical components necessary to make run-time probable GTV databases possible. Without being able to create our own GTV databases, we certainly could not have made a PPL database. For showing us the way, we are ever thankful.

References

Cash, M. and Miller, G. (2002). Checkers Solutions Bulletin Board Service, November 26. Since both MTC and PPL databases had been computed, G. Miller was offering a prize for the longest win found under the constraint that a Checker is unmoved for the duration.

Dodgen, G. and Trice, E. (2001). Co-authors of the WORLD CHAMPIONSHIP CHECKERS (WCC) program. An earlier, weaker version of the program (CHECKERS EXPERIMENTAL) was rated sixth in the world in 1992 (Schaeffer, 1997, p. 250), behind Dr. Marion Tinsley at number one, and the CHINOOK program at number two. See for more information about the current version of the program: http://www.WorldChampionshipCheckers.com.


Gilbert, E. (2002). KINGSROW checkers programmer. KINGSROW is currently ranked as the second strongest program in the world by virtue of placing second at the World Computer Checkers Championship held in Las Vegas, August 3-9, 2002.


Waldteufel, R. (2002). WYLLIE checkers programmer. July 21–27, 2002, Stonehaven, Scotland. WYLLIE defeated ACF 3-move World Champion Alex Moiseyev by the score of 16 wins, 3 losses, and 49 draws. Alex Moiseyev was also playing in the 2002 British Open tournament, where he finished in second place.

**Appendix A: Footnotes to Imperfect Moves Made**

1. 25-30 wins in 201, but 20-24 allows White 4 additional plies. Cumulative slip = 4 plies.
2. A rare case where the only move to win is a reversal of the previous move. This indicates that the GTV database must be consulted at every node in the tree during the search, a very expensive computation. Usually when one side is ahead by once piece, only the root of the tree needs to consult the database and prune the moves leading to draws and losses. The reasoning behind this is that in many cases, just about every move wins, so probing the database does not prune any legal moves, but it does slow down the search a great deal, even if the entire database is RAM-resident.
3. This retreat again is correct, but shuffling back and forth is usually penalized by an evaluation function. Notice the position is changing ever-so-slightly as the weak side has not shuffled back and forth over the same moves as the strong side. This is a very difficult position to play properly!
4. It should be noted that 24-20, again a repeated move on the strong side, would also lead to an optimal win in 189.
5. 19-16 wins in 187, but 19-23 allows White 4 additional plies. Cumulative slip = 8 plies.
6. Another instance where a reversing of the previous move is the only move to win. In the principal variation, the program expects 15-18, a non-optimal defensive move, instead of 15-11, the PPL best defense.
7. At ply 31, the program chooses this over its previous best candidate, 16-12, which was the optimal move. 16-12 wins in 185, but 16-20 allows White 2 additional plies. Cumulative slip = 10 plies.
8. 20-16 wins in 187, but 17-21 allows White 4 additional plies. Cumulative slip = 14 plies.
9. 16-12 wins in 185, but 16-19 allows White 2 additional plies. Cumulative slip = 16 plies.
10. 19-16 wins in 187, but 19-23 allows White 4 additional plies. Cumulative slip = 20 plies.
11. 19-16 wins in 187, but 19-24 allows White 4 additional plies. Cumulative slip = 24 plies. The program searched 63 plies and moved instantly for the strong side here, reporting a draw. This is because the hash table was saturated with positions consisting of only one move to win. The program will not make a drawing or losing move, but all of the moves maintaining the win have already been tried. On the strong side, the program tries not to repeat moves, but the weak side has created a position that will cycle in the hash table. This is the beginning of some serious trouble.
12. 24-20 wins in 189, but 25-30 allows White 6 additional plies. Cumulative slip = 30 plies.
13. 30-25 wins in 195, but 24-27 allows White 2 additional plies. Cumulative slip = 32 plies.
14. 30-25 wins in 195, but 24-27 allows White 2 additional plies. Cumulative slip = 34 plies.
15. 30-25 wins in 195, but 24-27 allows White 2 additional plies. Cumulative slip = 36 plies.
16. 30-25 wins in 195, but 24-27 allows White 2 additional plies. Cumulative slip = 38 plies. At this point, the program has slipped into a position that will repeat and allow the weak side to draw.
18. 17-14 wins in 179, but 31-27 allows White 4 additional plies. Cumulative slip = 8 plies.
19. 22-25 wins in 181, but 17-14 allows White 4 additional plies. Cumulative slip = 12 plies.
20. 14-17 wins in 183, but 22-17 allows White 4 additional plies. Cumulative slip = 16 plies.
21. 20-16 wins in 183, but 13-09 allows White 8 additional plies. Cumulative slip = 24 plies.
22. 09-13 wins in 185, but 09-06 allows White 4 additional plies. Cumulative slip = 28 plies.
23. 06-09 wins in 187, but 22-17 allows White 4 additional plies. Cumulative slip = 32 plies.
24. The program makes the correct move after searching 63 plies then moving instantly, reporting a score of draw for the strong side. This is the same phenomenon that was observed in the other position, and the repetition spiral is about to begin.

25. 21-25 wins in 181, but 17-14 allows White 4 additional plies. Cumulative slip = 36 plies.

26. 21-25 wins in 183, but 14-09 allows White 8 additional plies. Cumulative slip = 44 plies.

27. Another instance of hitting the limit of 63 plies of search due to hash table saturation. This is the correct move to win optimally but the program is reporting a draw from the repetitions.

28. Although this is the correct move for the optimal win, the program searched for over four times as long to reach ply 31 on this move than for the average of the previous moves.

29. 13-17 wins in 181, but 13-09 allows White 4 additional plies. Cumulative slip = 48 plies.

30. 17-22 wins in 175, but 19-24 allows White 4 additional plies. Cumulative slip = 52 plies.

31. 17-21 wins in 177, but 17-14 allows White 4 additional plies. Cumulative slip = 56 plies.

32. 14-17 wins in 179, but 19-23 allows White 4 additional plies. Cumulative slip = 60 plies.

33. The classic problem of "wandering Kings" is present here. Now 45 moves into the game, with 31 moves of regression, the program has only advanced 14 full moves towards the goal. This was, by far, the longest time required to complete a 31 ply, search at 27 minutes 16 seconds.

34. Making the correct move, and spending only 47 seconds to complete 31 plies of search in this instance.

35. 18-22 wins in 171, but 24-27 allows White 4 additional plies. Cumulative slip = 64 plies.

36. 22-25 wins in 173, but 22-17 allows White 4 additional plies. Cumulative slip = 68 plies.

37. 18-22 wins in 175, but 18-14 allows White 4 additional plies. Cumulative slip = 72 plies.

38. 14-18 wins in 177, but 19-23 allows White 4 additional plies. Cumulative slip = 76 plies.

39. 10-14 wins in 177, but 23-19 allows White 4 additional plies. Cumulative slip = 80 plies.

40. Another long search, indicative of opportunities to give up the draw appearing in the anticipated line of play. 10-14 wins in 179, but 08-11 allows white 8 additional plies. Cumulative slip = 88 plies.

41. 10-06 wins in 185, but 22-17 allows White 4 additional plies. Cumulative slip = 92 plies.

42. Making the correct move after a research on ply 31, replacing the "wandering" 06-02 move.

43. 14-17 wins in 175, but 14-10 allows White 4 additional plies. Cumulative slip = 96 plies.

44. 10-14 wins in 177, but 12-08 allows White 4 additional plies. Cumulative slip = 100 plies.

45. 10-14 wins in 179, but 08-11 allows White 8 additional plies. Cumulative slip = 108 plies.

46. 10-06 wins in 185, but 22-17 allows White 4 additional plies. Cumulative slip = 112 plies.

47. 10-06 wins in 185, but 10-07 allows White 4 additional plies. Cumulative slip = 116 plies.

48. 14-17 wins in 175, but 23-27 allows White 4 additional plies. Cumulative slip = 120 plies.

49. 22-25 wins in 173, but 12-08 allows White 8 additional plies. Cumulative slip = 128 plies.

50. 08-12 wins in 177, but 08-11 allows White 4 additional plies. Cumulative slip = 132 plies.

51. 14-17 wins in 179, but 22-17 allows White 4 additional plies. Cumulative slip = 136 plies.

52. 08-12 wins in 175, but 17-14 allows White 4 additional plies. Cumulative slip = 140 plies.

53. 14-17 wins in 175, but 14-10 allows White 4 additional plies. Cumulative slip = 144 plies.

54. 10-14 wins in 175, but 12-08 allows White 4 additional plies. Cumulative slip = 148 plies.

55. 10-14 wins in 179, but 08-11 allows White 8 additional plies. Cumulative slip = 156 plies.

56. 10-06 wins in 185, but 22-17 allows White 4 additional plies. Cumulative slip = 160 plies.

57. As was seen in the position at [42], here too the correct move was made after a research on ply 31, replacing the "wandering" 06-02 move.

58. Another correct move, played here at move 94, leads to the same position at move 60, which was 68 plies ago.

59. 14-17 wins in 175, but 14-10 allows White 4 additional plies. Cumulative slip = 164 plies. The position here at move 97 is the same as was seen at move 63, playing in the cycle from 68 plies ago. See note [43].

60. 10-14 wins in 177, but 12-08 allows White 4 additional plies. Cumulative slip = 168 plies. The position here at move 98 is the same as was seen at move 64, playing in the cycle from 68 plies ago. See note [44].

61. 10-14 wins in 179, but 08-11 allows White 8 additional plies. Cumulative slip = 176 plies. The position here at move 99 is the same as was seen at move 65, playing in the cycle from 68 plies ago. See note [45].

62. 10-06 wins in 185, but 22-17 allows White 4 additional plies. Cumulative slip = 180 plies. The position here at move 100 is the same as was seen at move 66, playing in the cycle from 68 plies ago. See note [46].

63. 10-06 wins in 185, but 10-07 allows White 4 additional plies. Cumulative slip = 184 plies. The position here at move 102 is the same as was seen at move 68, playing in the cycle from 68 plies ago. See note [47].
This paper describes the design and development of two world-class Lines of Action game-playing programs: YL, a three time Computer Olympiad gold-medal winner, and MONA, which has dominated international e-mail correspondence play. The underlying design philosophy of the two programs is very different: the former emphasizes fast and efficient search, whereas the latter focuses on a sophisticated but relatively slow evaluation of each board position. In addition to providing a technical description of each program, we explore some long-standing questions on the trade-offs between search and knowledge. These experimental results confirm the conclusions made by earlier researchers in the domain of chess, thus showing that the trends are not game-specific. In particular, we see diminishing returns with additional search depth, and observe that the knowledge level of a program has a significant impact on the results of such experiments.

Keywords: Lines of Action, search, knowledge

1. Introduction

One of the most important considerations when designing a strategic game-playing program is the trade-off between knowledge and search.

To decide on the best move continuation, programs typically perform a look-ahead search, evaluate the positions at the leaves of the search tree, and then propagate those values back to the root using the minimax principle. A program that uses a sophisticated but time-consuming board evaluation can more accurately determine the merit of each game-state visited, at the cost of sacrificing some of the look-ahead depth. Conversely, a program that uses a faster but less sophisticated board evaluation method can perform a deeper search, improving its short-term tactical ability. There is also compensation toward better knowledge, in that each additional level of search provides a more refined approximation of the value of each preceding position.

The trade-off between knowledge vs. search has spurred a considerable amount of research interest in the past, mainly for the game of chess (Schaeffer, 1986; Berliner et al., 1990; Junghanns and Schaeffer, 1997; Heinz, 2000). This
paper provides further insights using the game of Lines of Action (LoA, for short) as a new test-bed. LoA is tactically and strategically complex, and programs can employ many of the advanced search techniques and enhancements used by successful chess programs.

Two of the world’s strongest LoA programs were developed hand-in-hand at the University of Alberta (Billings, 2000). One program, YL, developed by Yngvi Björnsson, uses a very fast but somewhat restricted framework for board evaluation, allowing deep look-ahead. The other program, MONA, developed by Darse Billings, employs a relatively slow evaluation function resulting in shallower search, but with added features that provide a better assessment of each position encountered. This fundamental difference in design philosophy provides an opportunity to investigate the relative importance of knowledge versus search.

The main contributions of this paper are: (a) descriptions of the design and cooperative development process of two high-performance game-playing programs, and (b) experimental investigation of some general trade-offs between search and knowledge, using a domain that is different from chess, but belongs to the same class of games.

The next section briefly presents the rules of LoA and summarizes important strategic game concepts to be considered by programs. Section 3 describes some of the many benefits of the co-development process. Sections 4 and 5 provide detailed technical descriptions of YL and MONA, respectively. Section 6 provides empirical results and some knowledge versus search experiments using the two programs. Finally, Section 7 summarizes the content and states conclusions.

2. Lines of Action

Lines of Action was invented by Claude Soucie in the early 1960s, and was popularized by Sid Sackson in his book “A Gamut of Games” (Sackson, 1969). The simple, elegant rules are now presented, along with an overview of some of the important strategic concepts that a high-performance program might consider incorporating.

2.1 Rules

Objective. The object of the game is to move all of your pieces into a single connected group. Pieces may be connected diagonally or orthogonally. The leftmost diagram in Figure 1 shows the initial board layout.

Movement. Black moves first, and players alternate, moving one piece per turn. A piece may move horizontally, vertically, or diagonally. Along a given line, the distance a piece moves is the same as the total number of pieces (of both colours) on that line. You may jump over your own pieces, but not your
opponent’s pieces. You may land on and capture your opponent’s pieces, which are then removed from the game. You may not land on your own pieces.

2.2 Strategic Concepts

In chess and checkers, having more pieces than the opponent is highly correlated with winning, and this property outweighs all other factors in importance. In contrast, there is no single dominant feature in the assessment of LoA positions. As such, it is quite common to make concessions in one positional factor in order to strengthen another, and strong programs are frequently able to manage these trade-offs in order to maximize several features simultaneously.

We now list some of the important principles of LoA encountered during the development of MONA and YL.

Material. In LoA, there is no clear consensus on whether having extra material is advantageous, neutral, or detrimental. Since the goal of the game is to connect all of one’s pieces into a single group, having fewer pieces can require less work to fully coordinate them. In contrast, having more pieces might make it easier to form one large group, and might also enable better control of the board, preventing the opponent from connecting their pieces. It may be the case that having more pieces than the opponent only offers an indirect advantage, by increasing the value of other properties mentioned in this section. Those indirect advantages may exceed the added liability of managing the extra pieces, yielding a net positive effect for material advantage. However, since those other attributes are being measured separately, the weight assigned to material difference may be zero, or negative.

Mobility. As with many other board games, it is normally advantageous to have a position with many options and possible continuations. Increased mobility generally entails increased flexibility. Simply having many moves available can make it easier for a player to develop their own plans, interfere with the opponent’s plans, and defend against an opponent’s immediate threats. Moreover, several types of moves can be identified, such as: moves that capture a piece, moves toward or away from the center of the board, moves that connect our pieces, or moves that cut an opponent group. Each of the distinct move types
can then be evaluated differently. Having the move (i.e., being the next player to move in a given position) can also be treated as a distinct characteristic of the position. The value of this privilege depends on other positional properties, and generally increases toward the end of the game.

**Centrality.** In LoA, controlling the center of the board is very important. This can be accomplished by direct occupation of the more central squares, or by tactical counter-measures that prevent the opponent from occupying those squares. The center is particularly important in view of the standard starting position. Since each side must unite pieces from opposite sides of the board, seizing control of the center gains the shortest route to unification, while simultaneously interfering with the opponent’s connection. Having a bias toward centrality also has added pragmatic value, giving the program a sound high-level “plan” of bringing its pieces together in the middle of the board.

**Piece Coordination.** There are several identifiable concepts under the broad heading of piece coordination. Each of these is normally with respect to the pieces of the same colour. First, we say two pieces are connected if they are orthogonally or diagonally adjacent to each other. A group is a strongly connected subset of pieces (the object of the game being to form a single group). A program may designate a main group to be the largest group, or perhaps the most central group. The concept of connectivity can be measured as the number of pairwise connections between pieces; or the total number of groups; or the number of pieces that are not connected to the main group. The proximity or cohesion is a measure of distance or scatteredness of a player’s pieces. An outlier is an isolated piece (typically at the edge of the board) that needs to be brought into connection with or proximity of the main group, or the majority of like-coloured pieces.

**Obstructions.** An opponent’s piece or group of pieces may constitute an obstruction to connection. A piece may block one direction of movement of an enemy piece. A strong defensive formation is a blockade along the second rank or file, which greatly restricts the mobility of enemy pieces along the edge, and disconnects them from other like-coloured pieces. The effectiveness of an enemy blockade can be greatly reduced by having a foothold, which is a piece on the second rank that extends the edge group toward the center, and creates a defect in the blockade wall. The center diagram in Figure 1 shows a strong blockade for White along the top edge, and a White foothold (labeled ’1’) in Black’s blockade on the left.

**Mate Threats.** A mate threat is a threat to win the game on the next move, by connecting all pieces into one group. In general, mate threats are devastating in LoA, since the opponent typically must weaken their position considerably to answer the threat. Given the rather highly constrained movement options, this
will commonly lead to subsequent mate threats, until finally there is no adequate response. The rightmost diagram in Figure 1 shows an example position where a long sequence of mate threats secures Black a win. Since they frequently lead to forced winning sequences, it can be worthwhile to detect statically certain types of mate threats, and give a large evaluation bonus for each one present. This property of LoA also encourages special-purpose algorithms near the end of the game, such as threat-based search; or the proof-number techniques seen in Sakuta et al. (2002) and Winands, Uiterwijk, and Van den Herik (2002).

3. Co-development, Co-evolution

The development of both YL and MONA began in February of 2000. Since YL was expected to be clearly superior in terms of engineering and search speed, the author of MONA decided early on to focus on having superior knowledge in the form of a better-informed evaluation function. This turned out to be a fortuitous decision, as the contrasting styles of play enabled both sides to learn far more from friendly contests than would have otherwise been possible, and the progress of both programs was greatly accelerated as a result.

YL is based on a very fast framework, and its evaluation function is fully incremental, meaning that it has very little work to do at each leaf node. In contrast, MONA has a work-intensive evaluation function applied to each leaf node. Overall, YL is about 22 times faster than MONA in terms of positions processed per second. In compensation, MONA evaluates each position somewhat more thoroughly.

To build a high-performance search engine like YL, the philosophy is: “start fast and stay fast”, meaning that speed considerations and optimizations must be made at every stage of development. However, it can become increasingly difficult to make significant changes to the highly constrained architecture. In contrast, the design of MONA is basic and flexible, so new features can be added without difficulty. Since the evaluation function is already very costly, new attributes can be added that are rather expensive to compute, with only a minimal impact on overall speed performance. One might say “if you’re slow anyway, take advantage of it!”.

These fundamental differences in approach significantly enhanced the co-evolution of YL and MONA. A much broader range of positions were explored than would been seen with self-play matches, and critical weaknesses in each program were quickly revealed when playing into the other program’s strength. The advantages of cooperative development do not end there. Since evaluation features are easy to add and experiment with in MONA, the slower program could be used as a proving ground for new ideas. If certain properties prove to be extremely valuable, they could then warrant the more difficult changes in YL. One example of this cross-fertilization occurred with “footholds” (a piece
on the second rank that diminishes the effect of an opponent blockade, as shown in Figure 1, and discussed in Section 5). This feature turned out to be so valuable in practice that a special effort was made to detect similar patterns within the framework of YL’s fast evaluation function. The co-evolution worked in both directions. For example, games that MONA lost to YL due to short-term tactical errors suggested game-specific knowledge that could be added to reduce the risk associated with that weakness. As a result, some of MONA’s knowledge is designed to compensate directly for the shallower search.

The development of MONA and YL was greatly facilitated by two e-mail games played against Kerry Handscomb (one of the strongest LoA players in the world, having tied for first place in the 2000 e-mail championship). Early versions of MONA and YL combined efforts against him, choosing the move with highest average score. The lessons learned from those games, and Kerry’s commentary, resulted in major improvements to the evaluation functions of both programs. Kerry also wrote a series of informative articles on LoA for the magazine Abstract Games (Handscomb, 2003) (see issues 1–3, and others). Another valuable source of LoA domain knowledge was Dave Dyer’s (2003) excellent website for the game, which includes the game records of past e-mail championships. For other resources, see the MONA and YL webpage (Billings, 2000).

4. YL

This section describes the architecture of YL, including the underlying framework, the board-evaluation scheme, and the search algorithm.

4.1 Line Decomposition

The program evaluates board positions line by line — that is, each file, rank, and diagonal is evaluated independently. The score of a board position is the sum of the scores of its lines. The board is decomposed into 32 lines as shown in Figure 2. The first diagram pictures the 8 files, the second diagram the 8 ranks. The two remaining diagrams show the diagonals, which are paired to form 8-square-long diagonals, hereafter referred to as extended diagonals. This pairing is done to achieve a more compact representation. An extended diagonal is still considered as two distinct diagonals for evaluation purposes.

Evaluating the lines independently makes it possible to use a fast table look-up evaluation scheme to score the board during game play. For each line there are only \(3^8 = 6561\) possible different piece configurations. The total number of configurations \((32 \times 6561)\) is thus small enough to be evaluated beforehand. This evaluation is done at program startup and stored in tables residing in memory, called evaluation tables. The game is divided into three game phases
The program represents a board position internally using integers, one for each line. Each integer takes a value in the range 0 to 6560, representing the current piece configuration for the line. Piece configurations are mapped into integers as:

\[ s_8 \times 3^7 + s_7 \times 3^6 + \ldots + s_2 \times 3^1 + s_1 \times 3^0 \]

where \( s_i \) identifies the occupant of the \( i \)-th square on the line (\( \text{empty} = 0 \), \( \text{black} = 1 \), and \( \text{white} = 2 \)). These piece-configuration numbers are updated incrementally as moves are made on the board. Removing or adding a piece from/to a square affects only the configuration of the four lines intersecting the affected square. One benefit of this representation is that the piece-configuration numbers can be used directly as indices into the evaluation tables. Evaluating a board position is then simply a matter of looking up the merit of each line in the evaluation table. This evaluation is also done incrementally by keeping track of the current board score, and adjusting it by the evaluation differences of only the line configurations that changed during a move. This requires only a few table lookups. This efficient way of representing and evaluating boards is not new. Similar board representations are used by some high-performance Othello programs (Buro, 1999).

4.2 Evaluation Function

The main attraction of the aforementioned line-by-line evaluation scheme is its efficiency. On the down-side, the type of features that can be expressed within this framework are necessarily somewhat restricted. However, several important features can be measured precisely (including material balance, number of connections, and piece mobility), whereas other features must be approximated (such as proximity, obstruction, and blockage effectiveness). Where such approximations are not sufficient we use special non-line-based patterns.

Material. YL has a slight dislike for being up material, increasing as the game progresses. Note that this does not necessarily imply that it is bad to capture pieces. Rather, YL captures pieces only if it gives positional advantages.
Mobility. The program distinguishes between four types of moves (in decreasing order of importance): capture moves, moves that establish a connection, regular moves, and moves that disconnect own piece formation. Note that a single move can belong to more than one category. The merit of such a move is the combined merit from all relevant categories. Also, the side to move gets a constant bonus.

Centrality. A bonus is given for both direct and indirect center control; the more centralized a piece is the higher bonus it gets. Pieces sitting on the edge of the board are penalized, although somewhat less if they can move towards the center.

Piece coordination. YL measures connectivity by summing the number of neighbours of the same colour over all pieces on the board. It also measures line-wise piece proximity (how far apart pieces lie on a line). However, experience showed this measure to be insufficient on its own. Therefore, before the Computer Olympiad in 2002, a new non-line-based proximity feature was added. It keeps track of the area (number of squares) of the minimal bounding box needed to enclose pieces of each side; the smaller the box, the higher bonus a side gets. These boxes also encourage outliers to start gravitating toward the rest of the pieces. In Figure 3, the diagram on the right shows the bounding boxes for both sides.

The program has no notion of how many groups there are on the board except, of course, detecting the “single group” end-of-game condition. A single remaining group is detected by doing a breadth-first search over all neighbours of the same colour, starting with the piece last moved. If the number of pieces visited is equal to the number of pieces a player has, we know the pieces form a single group (in case of a capture move we need also to check end-of-game condition for the opposing side). Detecting the single-group condition initially slowed the program down significantly. However, by keeping two bitmasks for each side indicating which columns and rows its pieces occupy, we can cheaply check for necessary conditions of a single group being formed, in which case we then do the more expensive connection test. In practice, this trick eliminates most calls to the breadth-first search. These bitmasks are also used to efficiently keep track of the proximity bounding boxes.

Figure 3. Example of a blockage penalty, and proximity bounding boxes.
**Obstructions.** YL gives additional penalty to edge pieces that are fully or partially blocked by the opponent’s pieces. For example, in the diagram on the left in Figure 3 both white pieces are penalized. The number of penalty points is determined by the length of the blocked lines: piece 1 gets in total 11 penalty points (shown with an ‘1’) whereas piece 2 is only penalized by only 1 point (shown with a ‘2’). However, this scheme overestimates the penalty for blockades with footholds (the blockade is less effective because the edge pieces are connected to the outside via the foothold piece). A line-by-line evaluation scheme is unable to detect such situations. Footholds do occur frequently enough in practice to warrant a special treatment. Thus a special configuration pattern is used in YL that looks at the second file/rank in conjunction with the first, allowing the program to detect whether blocked edge pieces are connected to the outside and then scale down the blockade penalty appropriately.

YL also detects (along lines) how many opposite coloured pieces there are in between one own pieces, slightly penalizing such obstructions.

### 4.3 Evaluation Weights

Each line is evaluated using the aforementioned features that are then combined into a single line value using a linear function. A linear function was chosen somewhat arbitrary. In practice we could equally well have used a more complex non-linear function without sacrificing performance. However, we have not experimented with such alternatives.

Instead of hand-tuning the evaluation weights, we initially used a temporal-difference learning method for determining the relative importance of each evaluation feature. This was a sensible decision given the limited expert knowledge about the game, and allowed us to obtain a initial set of weights superior to what we could come up with by hand. A version of the program using the learned weights won the gold-medal at the Computer Olympiad held in London in 2000. However, further tuning of the weights (mainly based on observations from tournament play) and, in particular, introduction of new evaluation features have since then significantly increased the program’s playing strength.

### 4.4 Search

YL uses a traditional alpha-beta-based search algorithm (more specifically Principal Variation Search (Marsland, 1982)). The algorithm is augmented with many state-of-the-art enhancements, such as: iterative deepening, aspiration windows, a two-level transposition table, extensive automatically built opening book, repetition detection, and thinking on opponent’s time. The program also employs two well-documented speculative pruning schemes: null-move (Beal, 1989) and multi-cut pruning (Björnsson and Marsland, 2001).
As mentioned earlier, evaluation of game positions is done incrementally using table lookups. Move generation is also relatively fast, in part because legal moves for all possible line configurations are pre-calculated at program startup. The program employs both static (based on evaluation-table values) and dynamic (transposition-table move, killer-moves and history-heuristic) move-ordering techniques. A hierarchical move-ordering approach is used: in the upper levels of the tree (closer to the root) a more sophisticated move ordering is employed whereas at the lower levels a faster, although somewhat less sophisticated, ordering mechanism is used. All together, this results in a very fast and efficient search (the program typically explores close to 1.5 million positions-per-second in the opening on a 2.4GHz P4 PC).

5. Mona

This section describes the search engine and the evaluation function of Mona. As stated before, emphasis is on evaluation.

5.1 Search Engine Components

Mona is a fairly basic alpha-beta search program, using Principle Variation Search. The data structures for board representation, evaluation features, and move lists are simple integer arrays. Most of the well-established search enhancements are used, such as iterative deepening, transposition tables (with embellished Zobrist hashing), and the null-move heuristic.

Since move generation is used in the leaf-level evaluation function as well as the search process, the program spends a significant fraction of its execution time in this procedure. An optimization called move gathering was implemented, where all possible moves for each line configuration (row, column, or diagonal) are pre-computed, and those short move lists are concatenated at runtime. This resulted in a 40% speed-up to the program. A further 200-300% speed-up might be possible by incrementally carrying the move list indices (the new bottleneck), but this was not done prior to the program’s retirement in 2001.

Good move ordering is accomplished with a two-level hierarchy. First, the transposition table move is considered (i.e., the move which produced the best score the previous time the position was encountered, typically in the previous iteration). The second level is the default static move ordering, which ranks the general desirability of each move. A move toward the center of the board is rated higher (specifically, the centrality of the destination square), and capture moves are given a small bonus. Since this ranking is over a fixed interval (2-16), a linear time Radix sort is used to order the move list. This default move ranking is built into the move generator to reduce overhead. Move ordering with killer moves and the history heuristic are available as an option, but do not significantly increase program performance.
Since a prominent odd-even effect is observed in evaluations, and since mate threat detection (described below) only occurs on odd-ply searches, MO\(NA\) iterates 2-ply at a time. The courser granularity of iterative deepening results in an inefficient use of time when using a fixed-time-per-move time control, but this is partially offset by the savings from not doing even-numbered iterations.

5.2 Evaluation Function Components

The strength of MO\(NA\) lies in the evaluation function, which attempts to assess several properties of strategic importance, in the hope that this information will more than compensate for the overhead added by the relatively expensive computations.

The most basic evaluation simply determines whether the position is won, lost, drawn (by simultaneous connection), or unknown. This is done with a low-overhead breadth-first search to identify each group. If there is only one group of a given colour, then the game is over. The most important components of the full evaluation function are centrality, mobility, thickness, and mate threats. Useful refinements consider outlier mobility, blockades, footholds, outlier blocking, the progress toward connectivity, and the value of the move.

In most cases, it is the net difference between Black pieces and White pieces that is of interest. For example, the program would willingly reduce its own mobility provided that the opponent’s mobility is reduced by an even greater amount. For the most part the evaluation is symmetric (with the exception of outlier mobility, and mate threats).

Centrality. From a practical programming point of view, centrality is more than just a feature – it can be thought of as an overall game plan. The other two most important features in \(MO\(NA\)’s evaluation function (mobility and thickness) also have a significant centrality bias. This lends a degree of “harmony” to the evaluation, in that they are all striving for mutually supportive goals, rather than being at odds with each other. And indeed, it is quite common to sacrifice some of one commodity in order to gain more of another, in a cyclic process that eventually reaches positions that are powerful on all three counts.

To quantify centrality, each square is assigned a weight corresponding to the sum of orthogonal distances from the nearest corner (the four corner squares having a weight of 2, up to the four center squares having a weight of 8). The net centrality is relative to the number of pieces remaining. Thus, a few pieces on squares near the center would have a higher average centrality than a large number of pieces scattered about the board.

Mobility. A basic measure of mobility would simply count the number of moves that can be made. However, some moves are generally better than others. As noted above, the static move ordering value is determined by the destination centrality, and whether the move captures an enemy piece. By summing over
those values, the mobility function is naturally biased toward the moves that are likely to be useful. This is an absolute measure, so having more pieces, and thus more moves, is generally favourable.

An interesting consequence results from giving capture moves a greater weight. This actually discourages even exchanges, because having the option to make a capture has more value than actually making that capture. Thus, all else being equal, the program favours building up the pressure on key squares rather than releasing the tension through an even exchange. This has considerable practical value, especially in play against humans, because the computer program can handle the extra burden of many complex continuations much better than a human player. The chance of the opponent making a fatal error is thus increased in practice. This principle is highly analogous to the famous chess adage "the threat is stronger than the execution".

**Thickness.** In general, "clumping" of pieces is desirable, and there is some value in having redundant connections, preventing a group from being cut into two. However, we want to avoid having groups that are "too heavy", thereby reducing its own mobility.

The measure of connectivity used by MONA is called center-thickness, or simply thickness. A straightforward measure would count the number of pairwise adjacent pieces on the board. The embellishment used is to weight each of those pairs according to the centrality of the squares they occupy. This is another cumulative measure, so having more pieces is generally favourable. MONA uses a zero weight for the material factor, but still exhibits a preference for extra pieces, based on mobility and thickness.

It should be clear that the centrality biases built into mobility and thickness will generally encourage pieces to be moved away from the edge; and for groups to be formed in the center, if possible. However, this is not a heavy-handed bias, and cannot be easily obtained against quality opposition. It is simply a preference over other types of moves and piece formations. The relative weights of these three evaluation terms were set to have roughly equal contributions, with a slight preference for mobility, since it usually has a bit more pragmatic value. Very little was done in the way of tuning, and it is unknown if the program's performance could be enhanced significantly with more thorough experimentation.

**Mate Threats.** MONA computes many useful properties for each position. All groups are identified with the breadth-first search described previously. MONA designates the largest group to be the main group (choosing arbitrarily among equals).

Since the full move list is also available, it is possible to selectively do a check for the case where a single remaining outlier has a move that will put it adjacent to the main group, which constitutes an immediate threat to win the game.
This particular type of mate threat is the most common in practice, and can be detected without a full extra ply of look-ahead. While it is possible to write a special-purpose mate solver that looks only at direct threats and responses, we found that most of that utility was accomplished by simply knowing that a threat exists. MONA assigns a huge bonus for each such threat, which dominates all other evaluation terms. Thus it will always choose the best move among those that contain a threat (or highest number of multiple threats).

This policy is something of a gamble, since the threat may not actually lead to a win. However, to date we have only seen two cases where a winning position was lost by chasing a specious mate threat (both were against YL and, unfortunately, one in an important game).

**Outlier Mobility.** Given the rich information maintained about the board position, MONA can determine the individual mobility for every piece that is not part of the main group. She applies a non-linear penalty to the least mobile outlier for each side. Only the single worst outlier is considered. The search will naturally uncover combinations of moves that improve more than one outlier.

The weights for this feature are heavily skewed, making our worst outlier more important than the opponent’s worst outlier. The reasoning is that we do not want to invest a lot of energy (and evaluation points) on trying to trap an opponent piece that might easily escape, leaving us in a weakened position. Conversely, it is also risky to allow our own outliers to be trapped, since there may be no effective way to solve the problem (especially against humans, who can easily visualize such futures). The program is careful to avoid losing a game due to such traps, while preferring to build up steadily a strong position rather than trying to trap the opponent. This is one of several examples where the evaluation is consistent with the natural strengths and weaknesses of the program.

**Blockades and Footholds.** MONA’s evaluation function expends a lot of effort on the analysis of blockades along the second rank (see center Figure 1). These formations arise naturally from the standard starting position, even when using only the basic evaluation knowledge. The special purpose evaluation assesses the effectiveness of each blockade. Each additional blocking piece has a multiplicative effect on the penalty, while the presence of a foothold nullifies it almost completely. The program also distinguishes between a blockade of the main group and a blockade of outliers, with the latter being more serious.

**Other Features.** Since the character of LoA positions change radically during the course of the game, it is desirable to alter the overall plan and assessment to match the prevailing conditions. As a case in point, even the most refined positional evaluation is of little use in the final stage of the game – the only relevant question is whether we can form a single connected group before the opponent does. Empirically, it was found that the value of having the move
increases steadily toward the end of the game, when having the initiative is usually decisive.

The progress toward game completion is actually measured in a variety of ways. One of the simplest is to count the number of remaining pieces that are not part of the main group. A bonus is given for having two remaining outliers, and a larger bonus for having only one (since mate threats are commonly on the horizon). However, it is dangerous to try to converge too quickly, as it may fail tactically after all of the positional advantage has been sacrificed.

Many of the strategic properties perceived by MONA’s evaluation function cannot possibly be uncovered within a practical search horizon. For example, a piece trapped behind a wall of enemy pieces can be identified by static analysis, but the consequences of having that piece trapped might not begin to be felt until many moves in the future. MONA can add this type of knowledge without much down-side, whereas it is difficult to define within the framework of YL, and might be prohibitively expensive in any case (resulting in a net decrease in performance).

6. Empirical Results and Experiments

In this section we first provide insights into the playing strength of YL and MONA by reviewing tournament results. Secondly, we investigate the trade-offs of knowledge versus search in the game of LoA both via self-play and by matching the two programs against each other.

6.1 Over-The-Board Competitions

After a series of mutually beneficial friendly matches against each other, YL and MONA made their competitive debut in the University of Alberta Lines of Action Open, in April 2000 (Billings, 2002). The tournament was a double round-robin format with a time control of 20 seconds per move. YL won with a perfect 22-0 score, and MoNA finished second at 19-3.

In August 2000, both programs competed in the Fifth Computer Olympiad in London, England. YL won the gold medal, and MoNA won the silver. Although MoNA lost a critical game to YL on time only seconds before proving a win, both authors believed YL to be the stronger program at the 30-minute per game time constraints. The program MIA, by Mark Winands at University of Maastricht, took the bronze medal. YL successfully defended its title at the Sixth Computer Olympiad in 2001 ahead of MIA-II, and again won the gold medal at the Seventh Computer Olympiad in 2002 ahead of a steadily improving MIA-III (Björnsson and Winands, 2002). MoNA did not compete in either event. In July 2002, a four game friendly match was played between MIA-III and the original MoNA from 2000. Each program won two games, and based on further analysis of the moves played, it appeared that MIA-III had largely closed the gap that previously existed.
Table 1. MONA’s e-mail results against the top human players.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Rating</th>
<th>Colour</th>
<th>Name (Country)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2763</td>
<td>W</td>
<td>MONA (Canada) 25 wins, 0 draws, 0 losses</td>
</tr>
<tr>
<td>3</td>
<td>2374</td>
<td>W</td>
<td>Jorge Gomez Arrausi (Spain)</td>
</tr>
<tr>
<td>4</td>
<td>2202</td>
<td>WB</td>
<td>Claude Chaunier (France)</td>
</tr>
<tr>
<td>5</td>
<td>2192</td>
<td>BW</td>
<td>Kerry Handscomb (Canada)</td>
</tr>
<tr>
<td>6</td>
<td>2102</td>
<td>W</td>
<td>Uli Vogel (Germany)</td>
</tr>
<tr>
<td>7</td>
<td>2086</td>
<td>W</td>
<td>Ragnar Wikman (Finland)</td>
</tr>
<tr>
<td>8</td>
<td>2062</td>
<td>W</td>
<td>Hartmut Thordsen (Germany)</td>
</tr>
<tr>
<td>9</td>
<td>2037</td>
<td>W</td>
<td>Uli Vogel (Germany)</td>
</tr>
<tr>
<td>10</td>
<td>1999</td>
<td>W</td>
<td>Dave Dyer (USA)</td>
</tr>
<tr>
<td>11</td>
<td>1981</td>
<td>BB</td>
<td>Patrick Duff (USA)</td>
</tr>
<tr>
<td>12</td>
<td>1919</td>
<td>B</td>
<td>John Bosley (New Zealand)</td>
</tr>
<tr>
<td>13</td>
<td>1871</td>
<td>W</td>
<td>Fred Kok (Netherlands)</td>
</tr>
</tbody>
</table>

6.2 E-mail Correspondence Competitions

At deeper search depths, MONA’s strength increases dramatically. However, due to the expensive evaluation function, it can take a few hours to complete 11-ply early in the game, or 13-ply in the middlegame. This makes MONA particularly well-suited to e-mail correspondence play, with a pace of roughly one move per day.

Beginning in the summer of 2000, MONA began playing on Richard’s PBeM server (a popular play-by-email service) against many of the strongest known human players, winning every game played. MONA then won the Fifth Annual E-mail Tournament (the de facto world championship) with a perfect 14-0 record, including wins over most of the best LoA players in the world.

Table 1 lists some of the e-mail games played by MONA from May 2000 to May 2001. The chess-style ratings were calculated independently, using iterative re-computation over a database of more than 1000 PBeM LoA games until reaching convergence. The #2 rated correspondence player, at 2417, is the program MIA, which has only lost to the top human player, Jorge Gomez Arrausi. Jorge Gomez Arrausi won the 2000 e-mail championship, and was the top finishing human again in 2001, losing only to MONA. Several of the players listed are former LoA medalists at the Mind Sports Olympiad, including Fred Kok (gold twice), Hartmut Thordsen (gold), Ragnar Wikman (silver twice), and John Bosley (bronze).

MONA had the second move in most of the games against the top players, which is believed to be a larger disadvantage than having the Black pieces in chess. MONA also used considerably less time than her human opponents. In the final round of the 2001 e-mail tournament, MONA used an average elapsed time of 3.2 days per game, while her opposition used an average of 42.7 days per game against her. Based on the perfect record against elite competition, it is safe to conclude that the playing strength of MONA exceeds that of all human players by a considerable margin. However, it should be noted that LoA is still a young game, growing in popularity, and it is possible that “grandmaster”
calibre players could emerge in the future, giving programs a tougher challenge than has been seen to date. Programs will also continue to improve, and as they help humans to deepen their understanding of the game, that in turn could provide new knowledge to be added to future programs.

In April 2001, an 8-game match was played between older versions of MONA and YL, using the correspondence time control of 8 hours per move. Each game took roughly one week to complete. It is likely that these games constituted the highest level of play ever attained in LoA at that time. MONA won the match convincingly, with 7 wins and 1 loss. We intend to repeat this experiment using a more recent (and much stronger) version of YL.

6.3 Knowledge versus Search Experiments

In general, the farther a program looks ahead, the better it plays. This is the main justification for designing fast-searching game-playing programs. Historically, deeper search in chess programs has always led to significant improvements in performance. However, experimental studies have demonstrated diminishing returns with additional search depth (Junghanns and Schaeffer, 1997; Heinz, 2001).

We are also interested in investigating the importance of knowledge as LoA programs are given more time to think, since this will be a good predictor of how faster hardware platforms in the future will affect a program's playing strength. Traditionally, such investigations have involved a series of self-play experiments.

Constant Knowledge (Self-play) Experiments. First we repeated the most common self-play experiments, using YL and MONA. The results of those matches are shown on the left in Table 2. Each data point is the outcome of a 200-game match. A standardized set of 100 3-ply openings was defined (available upon request), and each player played both sides of each opening.

As in chess, searching deeply is obviously important: the deeper searching program invariably outperforms the shallower searching program by a considerable margin. However, as the search depth increases, the winning margin decreases, supporting the aforementioned experimental results found in the domain of chess.

Also of interest is that YL appears to benefit more from the deeper search than MONA. As noted in previous discussion, some of the knowledge in MONA

<table>
<thead>
<tr>
<th>Time(sec)</th>
<th>YL vs YL</th>
<th>YL vs MONA</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>77.50</td>
<td>81.00</td>
</tr>
<tr>
<td>8</td>
<td>79.75</td>
<td>79.25</td>
</tr>
<tr>
<td>32</td>
<td>83.25</td>
<td>80.25</td>
</tr>
<tr>
<td>128</td>
<td>81.00</td>
<td>65.75</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Depth</th>
<th>YL vs YL</th>
<th>MONA vs MONA</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 vs 7</td>
<td>89.75</td>
<td>79.50</td>
</tr>
<tr>
<td>7 vs 9</td>
<td>85.75</td>
<td>78.00</td>
</tr>
<tr>
<td>9 vs 11</td>
<td>79.75</td>
<td>72.50</td>
</tr>
<tr>
<td>11 vs 13</td>
<td>79.00</td>
<td>-</td>
</tr>
<tr>
<td>13 vs 15</td>
<td>72.75</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2. Fixed and variable knowledge experiments.
directly compensates for the shallower search; whereas YL must depend on the
deeper search to actually witness certain short-term tactics, refining its minimax
evaluation of the position. The results of 200-game matches alone are not
conclusive, but the same trend has been observed for other experiments as well.

**Varying Knowledge Experiments.** The self-play experimental setup only
shows the benefits of additional search when playing against a *very similar*
program. This does not address the possible effects of knowledge. The exper-
iments reported above might be misinterpreted to infer that YL would benefit
more than Mona from faster hardware – the exact opposite is true!

To test the knowledge differences directly, YL was also played against the
2001 version, labeled YL01. The more recent version was greatly improved
with the addition of several types of knowledge and evaluation features, but
the two YL programs are almost identical in search capacity, due to the pre-
calculated evaluation tables described earlier. Although Mona has not un-
dergone any significant changes since the 2000 Computer Olympiad, it is still
regarded to have the best-informed evaluation function of the three programs.

The results of those matches are shown to the right in Table 2. At shorter
time controls, YL outperforms both YL01 and Mona by a similar winning
margin. As the time controls get longer, YL continues to outperform YL01 at
a comparable win rate, indicating that the improvements in knowledge (the only
significant difference between the two programs) continue to pay dividends as
search depth increases.

However, the win rate against Mona drops off dramatically once the latter
is given at least two minutes per move, despite the fact that YL continues to
outsearch Mona by almost 3-ply on average. (This trend continues at longer
time controls, but those experiments were not complete and are not shown here). Presumably, the fast search engine is gaining less and less from deeper search, while the knowledge advantage continues to provide sustainable benefits.

Further evidence is seen in matches between Mona and YL with equal
fixed depths (using comparable sets of search enhancements). Whereas Mona
won about 55% of games at 5-ply, 7-ply, and 9-ply (54.50%, 55.25%, and
56.00%, respectively), her win rate increased to 71.00% when searching 11-ply
per move.

The implication of these observations is that faster hardware platforms do
indeed benefit knowledge-rich programs more than fast searchers. This in turn
suggests that when developing strategic game-playing programs, time invested
on improving the program’s board evaluation will generally pay greater div-
idends in the long run than effort spent on search improvements (especially
in view of the ever increasing difficulty in obtaining significant improvements
in search efficiency). Similar behaviour has been observed in chess programs
(Berliner et al., 1990).
7. Conclusions

We have revisited some long-standing questions regarding the roles of search and knowledge in high-performance game-playing programs, using two champion Lines of Action programs which emphasize different aspects of these contrasting approaches. Although the experiments are far from exhaustive, the results obtained so far are entirely consistent with previous studies for the game of chess. This supports the view that these are general phenomena, rather than game-specific.

By considering the effects of the knowledge level of game-playing programs over increasing search depths, it is possible to get a glimpse of what will likely be seen in the future. Although it is clear that search depth is and will continue to be very important, there are definite indications that increasing the knowledge holds the greater promise for lasting improvements in performance.

References

AN EVALUATION FUNCTION FOR LINES OF ACTION

M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk
Institute for Knowledge and Agent Technology, Department of Computer Science,
Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands
{m.winands,herik,uiterwijk}@cs.unimaas.nl, http://www.cs.unimaas.nl/m.winands/

Abstract Lines of Action (LOA) is a two-person zero-sum chess-like connection game. Building an evaluation function for LOA is a difficult task because not much knowledge about the game is available. In this paper the evaluation function of the tournament program MIA is explained. This evaluator consists of the following nine features: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player to move. These features have resulted in the evaluator MIA IV. The evaluator is tested in a tournament against other LOA evaluators, which have performed well at the previous Computer Olympiads. Experiments show that MIA IV defeats them with large margins. It turns out that the evaluator even performs better at deeper searches.

Keywords: Lines of Action, evaluation function, MIA

1. Introduction

LOA is a two-person zero-sum game with perfect information; it is a chess-like game with a connection-based goal, played on an 8 × 8 board. LOA was invented by Claude Soucie around 1960. Sid Sackson (1969) described it in his first edition of A Gamut of Games. After this publication, LOA received some attention of AI researchers. For instance, the first LOA program was written at the Stanford AI laboratory around 1975 by an unknown author. In the 1980s and 1990s “hobby” programmers wrote several LOA programs. However, all were beatable by humans (Dyer, 2000). At the end of the nineties LOA again became a target of AI researchers. Some of them used LOA only as a test domain for their algorithms, others tried to build strong LOA programs by using new ideas. The programs YL, MONA and MIA (Maastricht In Action) belong to the latter category. MIA finished third, second and again second at the fifth, sixth and seventh Computer Olympiad, respectively (Björnsson, 2000; Björnsson and Winands, 2001; Björnsson and Winands, 2002). The program can be played online at the website: http://www.cs.unimaas.nl/m.winands/loa/.
The standard framework of the $\alpha\beta$ search with its enhancements offers a good start for building a strong game-playing program. The real challenge in LOA is building a decent evaluation function, which incorporates the strategic intricacies of the game. The difficulty lies in the fact that knowledge about LOA evaluation functions is not well developed, although some material on this topic has been published (Winands et al., 2001). In this paper we discuss the latest evaluation function used in the program MIA.

The remainder of this paper is organised as follows. Section 2 explains the game of Lines of Action and describes the search engine. In Section 3 the evaluation function is explained. This evaluation function is tested against other evaluators in Section 4. Finally, in Section 5 we present our conclusions and topics for future research.

2. Test Environment

In this section we explain first the game of Lines of Action. Next, the search engine of MIA is described briefly.

2.1 Lines of Action

LOA is played on an 8 x 8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The players alternately move a piece, starting with Black. A move takes place in a straight line, exactly as many squares as there are pieces of either colour anywhere along the line of movement. A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit. The connections within the unit may be either orthogonal or diagonal. In the case of simultaneous connection, the game is drawn. If a player cannot move, this player has to pass. If a position with the same player to move occurs for the third time, the game is drawn.

Analysis of 2585 self-play matches showed an average branching factor of 29 and an average game length of 44 ply. The game-tree complexity and state-space complexity are estimated to be $O(10^{23})$ (Winands et al., 2001) and $O(10^{64})$, respectively. A characteristic property of LOA is that it is a converging game (Allis, 1994), since the initial position consists of 24 pieces, and during the game the number of pieces (usually) decreases. However, since most terminal positions have still more than 10 pieces remaining on the board (Winands, 2000), endgame databases are (probably) not effectively applicable in LOA. As a case in point, we remark that an endgame database of ten pieces would require approximately 10 terabytes. Finally, in LOA the standard chess notation for moves is used.
2.2 MIA’s Search Engine

MIA performs an $\alpha\beta$ depth-first iterative-deepening search. Several techniques are implemented to make the search efficient. The program uses PVS (Principal Variation Search) to narrow the $\alpha\beta$ window as much as possible (Marsland and Campbell, 1982). A two-deep transposition table (Breuker et al., 1996) is applied to prune a subtree or to narrow the $\alpha\beta$ window. At all interior nodes which are more than 2 ply away from the leaves, the program generates all the moves to perform the Enhanced Transposition Cutoffs (ETC) scheme (Schaeffer and Plaat, 1996). Next, a null move (Donninger, 1993) is performed before any other move and it is searched to a lower depth (reduced by $R$) than other moves. In the search tree we distinguish three types of nodes, namely PV nodes, CUT nodes, and ALL nodes (Knuth and Moore, 1975; Marsland and Campbell, 1982). The null move is done at CUT nodes and at ALL nodes. At a CUT node a variable scheme, called adaptive null move (Heinz, 1999), is used to set $R$. If the remaining depth is more than 6, $R$ is set to 3. When the number of pieces of the side to move is lower than 5 the remaining depth has to be more than 8 for setting $R$ to 3. In all other cases $R$ is set to 2. For ALL nodes $R = 3$ is used. If the null-move does not cause a $\beta$-cut, multi-cut (Björnsson and Marsland, 1999) is performed. Experiments showed that using multi-cut is not only beneficial at CUT nodes but also at ALL nodes (Winands et al., 2003). For move ordering, the move stored in the transposition table, if applicable, is always tried first. Next, two killer moves (Akl and Newborn, 1977) are tried. These are the last two moves, which were best or at least caused a cut-off at the given depth. Thereafter follow: (1) capture moves going to the inner area (the central $4 \times 4$ board) and (2) capture moves going to the middle area (the $6 \times 6$ rim). All the other moves are ordered decreasingly according to their scores in the history table (Schaeffer, 1983). In the leaf nodes of the tree a quiescence search is performed. This quiescence search looks at capture moves, which form or destroy connections (Winands et al., 2001) and at capture moves going to the central $4 \times 4$ board.

3. Evaluation Function

In this section the evaluation function of MIA is explained. This evaluator consists of the following nine features: concentration, centralisation, centre-of-mass position, quads, mobility, walls, connectedness, uniformity, and player to move. These features are described below in detail (Subsection 3.1 to 3.9), followed by some information about the use of caching (Subsection 3.10).
3.1 Concentration

The concentration of the pieces is calculated by a centre-of-mass approach. In MIA this is done in four steps. First, the centre of mass of the pieces on the board is computed for each side. Second, we compute for each piece its distance to the centre of mass. The distance is measured as the minimal number of squares from the piece to the centre of mass. These distances are summed together, called the sum-of-distances. Third, the sum-of-minimal-distances is looked up in a table. It is defined as the sum of the minimal distances of the pieces from the centre of mass. This number is necessary since otherwise boards with a few pieces would be preferred. For instance, if we have ten pieces, there will be always eight pieces at a distance of at least 1 from the centre of mass, and one piece at a distance of at least 2. In this case the sum-of-minimal-distances is 10. Thus, the sum-of-minimal-distances is subtracted from the sum-of-distances, the result being called the surplus-of-distances. Fourth, we calculate the concentration, defined as the inverse of the average surplus-of-distances. Since by doing so we reward positions with pieces in the neighbourhood of each other, eventually they will be connected in solid formations or they will create threats to win.

3.2 Centralisation

Each piece gets a value dependent on its board square according to this feature. Pieces at squares closer to the centre are given higher values than the ones farther away. Pieces at the edge are given a negative value. This is done because such pieces are easy to block by a wall (see Subsection 3.6). Pieces at the corner are punished even more severely. To prevent the program from over-aggressively capturing pieces, the average is computed instead of the sum of piece values.
3.3 Centre-of-mass Position

In earlier versions of MIA positions with a somewhat more centralised centre-of-mass were slightly preferred. The idea was to prevent formations from being built on the edges, where they are more easily destroyed or blocked. Interestingly, after applying Temporal-Difference (TD) learning the weight for the centralised centre-of-mass feature is changing its sign (Winands et al., 2002), which means that opposite to expectations it is good to have the centre-of-mass closer to the edge instead of in the centre. If the centre-of-mass is in the centre, it is possible that pieces are scattered over the board (e.g., the white pieces in Figure 1a). If the centre of mass is at the edge, pieces have to be in the neighbourhood of each other, otherwise they would lie outside the board. Another plausible explanation of why it is worse to have the main piece formation in the centre is that it can be more easily attacked there, whereas groups residing closer to the edge can only be attacked from one side.

3.4 Quads

The use of quads for a LOA evaluation function was first proposed and implemented by Dave Dyer in 1996 in his program LoAJava and empirically evaluated by Winands et al. (2001). This feature counts certain quads types. A quad is defined as a $2 \times 2$ array of squares (Gray, 1971). In this feature we only consider quads of three (Q$_3$) or four pieces (Q$_4$) of the same colour, since it is impossible to destroy these formations by a single capture. However, the danger exists that many of those quads are created outside the neighbourhood of the centre of mass. So, in MIA we have rewarded only Q$_3$'s and Q$_4$'s, which are at a distance of at most two of the centre of mass. For instance, Black has two Q$_4$'s in Figure 1b.

3.5 Mobility

In the mobility feature the number of moves for each side are computed. This feature was first implemented in Mona and YL. In previous evaluation functions of MIA all moves were weighted equally. However, experiments have shown that certain move types are better than others (see also Hashimoto et al., 2003). Therefore, in MIA the following bonus/malus system is applied: the value of a capture move is doubled, the value of a move going to an edge or a move along an edge is halved. If a move belongs to multiple categories, the bonus/malus system is used multiple times. For example, let us assume that a regular move gets value 1, then a capture move gets value 2, a capture move going to an edge gets value 1, a capture move in an edge line going to a corner gets value 0.5. The computational requirements of this component are not high. For each line configuration of pieces (represented as a bit vector) the mobility
can be precomputed and stored in a table. During the search, the index scheme can be updated incrementally and in the evaluation function only a few table lookups have to be done.

3.6 Walls

Because a piece is not allowed to jump over the opponent's pieces, it can happen that the piece is blocked, i.e., cannot move. Blocking a piece far away from the other pieces is an effective way of preventing the opponent to win. Even partial blocking, called a wall (Handscomb, 2000), can be quite effective, since it forces a player to find a way around the wall. Detecting whether a piece is (partially) blocked can be expensive as we have to know what the moves of the piece are and what its goal is. In MIA we look only at walls that prevent the opponent's edge pieces from moving toward the centre. These walls are quite common and effective. The patterns can be precomputed and therefore are easy to detect. For example, in Figure 2a the piece on a4 is blocked in three ways going to the centre, whereas the piece on h4 is only blocked in two centre directions. In the evaluator, we distinguish between walls which block two or three centre directions. We also remark that we take special care of walls which block corner pieces. For example, the piece on h8 is blocked only in two directions, but we evaluate this position as if it was blocked in 3 centre directions. The totally isolated piece on a8 is evaluated as if there were two walls which both block the piece in three directions. We only look at certain blocking patterns for edge pieces. For example, the pieces on b1 and c1 are completely blocked, but we take only the two 3-centre directions blocks into account. It is a subject of future research to incorporate more of these kind of patterns.

3.7 Connectedness

Although the concentration component and quad component favour solid formations in the centre, there is still room for a component which determines the connectedness of a side. In MIA we compute the average number of connections of a piece. In some evaluation functions the total number of connections is taken into account (e.g., YL), but this could implicitly be a material advantage. Any kind of material component in LOA evaluation functions is always tricky because the program might wildly capture pieces. This feature does not take into account whether a connection is important or not. To distinguish this, a global look at the board would be needed, which is time consuming. The number of connections for each side in each line configuration can be precomputed as is done with the mobility component.
3.8 Uniformity

The disadvantage of the centre-of-mass approach is that it aims to connect as many pieces as possible in a local group, hardly worrying about some remote pieces (orphans). It is sometimes hard to connect these orphans. For instance, in Figure 2b the black pieces are grouped around e2, but the black piece on b8 is rather far away from this group. To prevent that one or more pieces become too remote from the main group, a feature is used which aims at a uniform distribution (Chaunier and Handscomb, 2001) to counterbalance the negative effects of the centre-of-mass approach. In our program this is done in a way which is primitive but effective. The area of the distributed pieces is computed, assuming it is a rectangle. The smaller the area is, the higher the reward is. An analogous implementation was first done in the program YL, but details are not known.

3.9 Player to Move

In the search tree not every leaf node has the same player to move. A small bonus is given to the moving side, since having the initiative is mostly an advantage in LOA (Winands, 2000) like in many other games (Uiterwijk and van den Herik, 2000).

3.10 Caching

It is possible in our evaluation function to cache computations of certain features, which can be used in other positions. Let us assume that we investigate the move b8-c8 in Figure 2b and evaluate the resulting position. If we next investigate b8-b7 we notice that certain properties of White’s position remain the same (e.g., the number of pieces, centre-of-mass, the number of connections), whereas others can change (e.g., moves, blockades). It easy to see that
we do not have to compute the concentration, centralisation, position of the centre-of-mass, quads, connection, and uniformity for White again. Evaluation of components, which are not dependent of the position of the other side, are stored in the evaluation cache table. In the current evaluation function this gives a speed-up of at least 60 percent in the number of nodes investigated per second.

4. Experiments

In order to quantify the improvements of the evaluation function, we played a round-robin tournament in which evaluators from earlier tournament versions of the program participated. All evaluators used the current search engine, described in Subsection 2.2. The evaluators are explained in Subsection 4.1. The results are described in Subsection 4.2.

4.1 Benchmark Evaluators

The benchmark evaluation functions are described below.

Evaluator: MIA I The core of this evaluation function is the centre-of-mass approach. The quad feature is also implemented. Pieces at the edge are given a negative bonus. Contrary to MIA IV a bonus is given for a centralised centre-of-mass (Winands et al., 2001). The weights of the features were carefully hand-tuned. In retrospect this evaluator was primitive, although it won a game against both MOnA and YL at the fifth Computer Olympiad (Björnsson, 2000).

Evaluator: MIA II The major change of this evaluation function compared to the previous one is the introduction of the mobility component. There is no discrimination in rewarding different move types. In this evaluator pieces at a corner edge are punished more severely. Using this evaluator the tournament program shared the first place with YL in the regular tournament at the sixth Computer Olympiad. The play-off match was won by YL (Björnsson and Winands, 2001).

Evaluator: MIA III This evaluation function is enhanced with the wall feature. The centralisation feature is improved by rewarding pieces in the centre. A bonus is given for the player to move. The major improvement was retuning all the weights by using TD-learning (Winands et al., 2002). There were three major changes in the weights. First, the initial weight of the dominating centre-of-mass was decreased to one tenth of its original value, indicating that we had overestimated the importance of this feature. Second, the weight for the centralised centre-of-mass feature changed its sign, which means that opposite to expectations it is good to have the centre-of-mass closer to the edge instead of in the centre. Third, the weight of the centralisation component increased the most, indicating that we had overestimated the importance of this feature. Using this evaluator the tournament program finished second at the seventh Computer Olympiad (scoring 1.5 points out of 4 against the much improved
winner YL) (Björnsson and Winands, 2002). An exhibition match was played against MONA during the Third International Conference on Computers and Games 2002 (CG'02), which ended in a 2-2 tie (Billings and Björnsson, 2002).

**Evaluator: MIA IV** This evaluation function incorporates all features as described in Section 3. The centralisation, wall, and player-to-move features used the same weights as the ones in MIA III. All the weights of the other features were basically found by using TD-learning. Some of them were adjusted by hand afterwards.

An overview of the separate features as used in the four evaluators is given in Table 1. Note that the weights and details of the features may differ between different evaluators.

<table>
<thead>
<tr>
<th>Feature</th>
<th>MIA I</th>
<th>MIA II</th>
<th>MIA III</th>
<th>MIA IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concentration</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Centralisation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>C.o.m. position</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Quads</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Mobility</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Walls</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Connectedness</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Uniformity</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Player to move</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1. Overview of the features.*

### 4.2 Results

The evaluators, previously described, played 1000 matches against each other in a round-robin tournament. They started always from the same 10 positions given in the Appendix, playing with both colours. To prevent that programs played the games over and over again, a sufficiently large random factor was included in each evaluation function.

Fixed-depth searches were used as time control instead of time. At first sight it may look as if we are favouring the more advanced evaluators (i.e., they are time intensive because of the extra knowledge). This is not a problem for two reasons. First, the difference in speed is quite moderate. The program runs only 15 per cent slower with the MIA IV evaluator than with the MIA I evaluator. All the evaluators have to compute the average distance to the centre-of-mass and the quads, which is time consuming. Most other additions are relatively cheap. Second, when an evaluator is a good predictor of the situation, a best move found at a shallow search is more likely to stay good and therefore causing cut-offs at deeper searches. For example, when the MIA I evaluator is used in the current search engine it searches 75 per cent more nodes compared to the
MIA IV evaluator. The advantage of fixing the depth is that we can measure the influence of increasing the depth.

<table>
<thead>
<tr>
<th>Evaluator</th>
<th>MIA I</th>
<th>MIA II</th>
<th>MIA III</th>
<th>MIA IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIA I</td>
<td>0</td>
<td>259</td>
<td>199</td>
<td>71.5</td>
</tr>
<tr>
<td>MIA II</td>
<td>741</td>
<td>0</td>
<td>373</td>
<td>163.5</td>
</tr>
<tr>
<td>MIA III</td>
<td>801</td>
<td>627</td>
<td>0</td>
<td>248.5</td>
</tr>
<tr>
<td>MIA IV</td>
<td>928.5</td>
<td>836.5</td>
<td>751.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2. Tournament results at depth 4.

<table>
<thead>
<tr>
<th>Evaluator</th>
<th>MIA I</th>
<th>MIA II</th>
<th>MIA III</th>
<th>MIA IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIA I</td>
<td>0</td>
<td>188</td>
<td>168.5</td>
<td>51</td>
</tr>
<tr>
<td>MIA II</td>
<td>812</td>
<td>0</td>
<td>356</td>
<td>174</td>
</tr>
<tr>
<td>MIA III</td>
<td>831.5</td>
<td>644</td>
<td>0</td>
<td>223.5</td>
</tr>
<tr>
<td>MIA IV</td>
<td>949</td>
<td>826</td>
<td>776.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3. Tournament results at depth 6.

<table>
<thead>
<tr>
<th>Evaluator</th>
<th>MIA I</th>
<th>MIA II</th>
<th>MIA III</th>
<th>MIA IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIA I</td>
<td>0</td>
<td>137</td>
<td>159.5</td>
<td>41.5</td>
</tr>
<tr>
<td>MIA II</td>
<td>863</td>
<td>0</td>
<td>360</td>
<td>129</td>
</tr>
<tr>
<td>MIA III</td>
<td>840.5</td>
<td>640</td>
<td>0</td>
<td>205</td>
</tr>
<tr>
<td>MIA IV</td>
<td>958.5</td>
<td>871</td>
<td>795.0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4. Tournament results at depth 8.

<table>
<thead>
<tr>
<th>Evaluator</th>
<th>MIA I</th>
<th>MIA II</th>
<th>MIA III</th>
<th>MIA IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIA I</td>
<td>0</td>
<td>97.5</td>
<td>137.5</td>
<td>44.5</td>
</tr>
<tr>
<td>MIA II</td>
<td>902.5</td>
<td>0</td>
<td>359.5</td>
<td>121.5</td>
</tr>
<tr>
<td>MIA III</td>
<td>862.5</td>
<td>640.5</td>
<td>0</td>
<td>234.5</td>
</tr>
<tr>
<td>MIA IV</td>
<td>955.5</td>
<td>878.5</td>
<td>765.5</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5. Tournament results at depth 10.

In Tables 2-5 the results of the tournaments are given for searches to depth 4, 6, 8, and 10, respectively. MIA IV defeats the previous evaluators of MIA with ease. Even the strong MIA III is not able to score more than 20 to 25 percent of the points against MIA IV. Although MIA II's only major improvement is a primitive mobility component, it did not only outperform MIA I, but it also played much better against MIA III and IV than MIA I did. Interestingly, the weak MIA I performs worse at deep searches, whereas the opposite holds for the strong MIA IV evaluator. A reason might be that at the one hand a deep search is not able to compensate the lack of knowledge of MIA I, while at the other hand a deep search exploits more of the potential of MIA IV.
5. Conclusions and Future Research

In this paper we have seen that MIA IV defeats the older evaluators by large margins. Most additions of MIA IV in knowledge are quite simple to evaluate and lead to big rewards in playing strength. It turns out that MIA IV even performs better at deeper searches.

More patterns of blocked pieces, better distinction of move types in the mobility component, and additional knowledge whether a connection is important are some of the issues which could improve the evaluator. There is still room to fine tune certain weights and parameters in the evaluation function. Until now the authors of the strong programs YL and Mona have not published the details of their programs’ evaluators. If their knowledge becomes available, combining their ideas with MIA IV would probably further increase the playing strength significantly.

Acknowledgements

The authors would like to thank Yngvi Björnsson and Darse Billings for sharing their thoughts about LOA in general, and LOA evaluation functions in particular. We also thank the anonymous referees for their valuable comments.

References


**Appendix: Start Positions**

Below the positions are given, which are used in the experiments of Section 4.
SOLVING 7×7 HEX: VIRTUAL CONNECTIONS AND GAME-STATE REDUCTION

R. Hayward, Y. Björnsson, M. Johanson, M. Kan, N. Po, J. van Rijswijck
Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada
{hayward,yngvi,johanson,mkan,nathan,javhar}@cs.ualberta.ca, http://www.cs.ualberta.ca/

Abstract We present an algorithm which determines the outcome of an arbitrary Hex game-state by finding a winning virtual connection for the winning player. Our algorithm performs a recursive descent search of the game-tree, combining fixed and dynamic game-state virtual connection composition rules with some new Hex game-state reduction results based on move domination. The algorithm is powerful enough to solve arbitrary 7×7 game-states; in particular, we use it to determine the outcome of a 7×7 Hex game after each of the 49 possible opening moves, in each case finding an explicit proof-tree for the winning player.

Keywords: Hex, virtual connection, pattern set, move ordering, move domination, game-state reduction

1. Introduction

Hex is the classic two-player board game invented by Piet Hein in 1942 and independently by John Nash around 1948 (Gardner, 1959; Nasar, 1998). The board consists of a rhombus-shaped \( n \times n \) array of hexagons, also called cells. Each player is assigned a set of stones and two opposing board sides, all with the same colour; say Black gets black stones and sides, while White gets white stones and sides. Players alternately place a stone on an unoccupied cell. The first player to form a path connecting his/her two sides with his/her stones wins the game. See Figure 1. For more on Hex, see Browne (2000) and Hayward and Van Rijswijck (200x).

In Hex, an unrestricted opening allows the first player to gain a considerable advantage: it is known that there exists a winning strategy for the first player (Gardner, 1959), and while no explicit strategy which holds for arbitrary sized boards is known, most players believe that opening in the centermost cell in particular is a winning move. In order to offset this opening move advantage, the game is often started according to the following “swap rule”: colours are assigned to the four sides of the board, but not to the players; one player then places a stone on any cell; the other player then chooses which colour stones
to play with. The second move is played by the player whose stones are the opposite colour of the first stone. From then on, the game continues in normal fashion, namely with players alternating moves.

With respect to Hex, a board-state describes a particular placement of some number of black stones and some number of white stones, such that each cell has at most one stone. We assume no constraint on the relative number of stones of each colour, as the game may have started with a handicap advantage for one of the players. The empty board-state has no stones on the board. A k-opening is a board-state with exactly k stones on the board. A turn-state describes which player has the next move. A game-state, or simply a state, consists of a board-state and a turn-state. We denote by $G = [P, B]$ the game-state with turn-state $P$ and board-state $B$; for this game-state, we say that $P$ wins $G$ if $P$ has a winning strategy for $G$. For a board-state $B$, we say that $P$ wins $B$ if $P$ wins $G = [P, B]$.

A state is solved if the winning player is known, and explicitly solved if a winning strategy is known. As we have already remarked, for arbitrarily large boards, Hex has been solved for the empty board-state, but not explicitly solved.

In this paper we consider the problem of solving arbitrary Hex states, and present an algorithm which solves this problem. The worst-case running time of our algorithm is exponential in the number of cells in the board, which is not surprising given that solving arbitrary Hex states is PSPACE-complete (Reisch, 1981). As a benchmark for the efficiency of our algorithm, we solve all 7×7 1-openings. Previously known 1-opening results are summarized in Figure 2.

Our results yield the first computer solution of any Hex state on a 7×7 or larger board. Solving Hex states on 5×5 or smaller boards is a computationally routine task. To solve arbitrary 6×6 Hex states, Van Rijswijk (2000, 1999-2003) used an alpha-beta search guided by a Hex-specific evaluation function; his algorithm solved all 1-openings and many longer openings. As this method was not strong enough to solve 7×7 states, he further described but did not implement an alternative recursive-descent algorithm (Van Rijswijk, 2002). Recently Yang et al. solved by hand several 7×7 1-openings (Yang et al., 2001, 2002a), one 8×8 1-opening (Yang et al., 2002b), and one 9×9 1-opening (Yang, 2003).
Solving $7\times 7$ Hex: Virtual Connections and Game-State Reduction

Figure 2. Previously known 1-opening results. The stone on each cell indicates the winner with perfect play if White’s first move is to that cell. For cells with no stone, the winner was not previously known. The $6\times 6$ results were obtained by Van Rijswijck by computer (Van Rijswijck, 2002). The $7\times 7$ results were obtained by Yang et al. (2001, 2002) by hand.

Our algorithm solves an arbitrary Hex state by computing a winning virtual connection according to dynamic-state composition rules. Following the recursive descent game-tree search proposed by Van Rijswijck, our algorithm is enhanced by the computation of fixed-state virtual connections; additionally, some new Hex move domination and state reduction results allow significant pruning of the game-tree.

Before presenting our algorithm in Section 4 and our $7\times 7$ results in Section 5, we provide necessary background information on virtual connections in Section 2 and state reductions in Section 3.

2. Connection Sets

Roughly, a connection set in Hex is a subgame in which one of the players can form a connection between two specified sets of cells. If the player can connect the two sets even if the opponent moves first, the connection set is called a virtual connection or link; if the player must have the first move in order to guarantee the connection, the connection set is called a weak connection or prelink.

More formally, with respect to a fixed Hex state, a player $P$, sets of cells $X, Y$, and a set of cells $S$, $(P:X, S, Y)$ is a virtual connection or link if there exists a strategy whereby, in the game restricted to the set of cells $X \cup S \cup Y$, $P$ can form a chain connecting at least one cell of $X$ with at least one cell of $Y$, even if $P$’s opponent moves first; in other words, $(P:X, S, Y)$ is a virtual connection if there exists a second-player-win strategy for $P$ to connect $X$ and

---

1Here each of the four sides is also be considered as a cell.
Figure 3. A virtual connection formed by weak connections. Each of the three leftmost figures shows a Black weak connection, indicated by the dotted cells, from the black stone to the bottom right side; the white dot indicates a cell whose occupation would transform the weak connection into a virtual connection. The common intersection of these weak connections is empty, so their union forms a Black virtual connection from the black stone to the bottom right side, shown in the rightmost figure.

In this paper, all virtual and weak connections have the form \((P:X, S, Y)\) where \(X\) and \(Y\) each consist of a single cell; we denote such connections \((P:x, S, y)\) where now \(x\) and \(y\) represent single cells instead of sets of cells.

Although defined slightly differently by different authors, virtual connections have long been recognized as being central to Hex strategy. References to virtual connections permeate the Hex literature, where they are also referred to as “connections” or “safe groups”. For example, virtual connections are discussed by Berge (1977)\(^2\) and Browne (2000).

Virtual connections are useful in solving states since, when accompanied by an explicit strategy, a virtual connection serves as a proof or certificate that a pair of cells can be connected.

In particular, if \(P\) has a virtual connection \((P:x, S, y)\) where \(x\) and \(y\) are the two sides belonging to \(P\), then this virtual connection certifies that \(P\) wins the game. For this reason, we call \((P:x, S, y)\), a win-link (respectively win-prelink) if it is a link (prelink) and \(x\) and \(y\) are \(P\)’s two sides. Since the sides of each player are fixed, we will sometimes abbreviate \((P:x, S, y)\) by \(P:S\) whenever \(x, y\) are the sides of \(P\).

Connection sets are particularly effective in Hex end-game analysis. For example, the following is a restatement in our terminology of an observation made by Berge.

---

\(^2\)A translated version of appears in Hayward (2003a).
THEOREM 1 (Berge, 1977; Hayward, 2003a) Consider a state in which a player $P$ has the next turn and $P$’s opponent $Q$ has one or more win-prelinks. Then $Q$ wins unless $P$’s next move is to a cell which intersects all $Q$-win-prelinks, for otherwise $Q$ can on the next move convert a win-prelink to a win-link.

In light of this result, for any fixed state and a player $P$ with opponent $Q$, we refer to the set of unoccupied cells in the intersection of all $Q$-win-prelinks as $P$’s mustplay region. Notice that the computation of a mustplay region is a form of null move analysis, as it involves the consideration of what can occur if a player skips a turn.

A useful feature of virtual connections is that smaller ones can be combined in various ways to form larger ones. The knowledge of this fact is as old as Hex itself; for example, it is discussed in detail by Berge (1977) and Hayward (2003a). Recently, Anshelevich (2002) used the following set of combining rules to compute connection sets in an inductive or “bottom-up” fashion. A $P$-stone is a stone belonging to $P$; $\emptyset$ denotes the empty set.

THEOREM 2 (Anshelevich, 2002) $(P:x, \emptyset, y)$ is a virtual connection if $x$ and $y$ are adjacent. Also, if $(P:x, S, y)$ and $(P:y, T, z)$ are virtual connections and $\{x\} \cup S$ and $T \cup \{z\}$ do not intersect, then $(P:x, S \cup \{y\} \cup T, z)$ is a virtual connection if $y$ is occupied by a $P$-stone and a weak connection if $y$ is unoccupied. Also, if $(P:x, S_1, y), (P:x, S_2, y), \ldots, (P:x, S_k, y)$ are weak connections and the common intersection of the sets $S_j$ is empty, then $(P:x, S, y)$ is a virtual connection, where $S$ is the union of the sets $S_j$.

Notice that this set of rules is static, in that it yields a class of connection sets for a fixed state. This set of rules is not sufficient to establish all virtual connections of a state, and is thus not strong enough to solve all Hex states. However, the rules do yield a sufficiently large class of virtual connections to provide an effective subroutine of a strong Hex-playing program (Anshelevich, 2002).

As Van Rijswijck observed, an alternative method of computing connection sets is to proceed through the game-tree dynamically. Let $G = [P, B]$ be a state and let $Q$ be the opponent of $P$. For each unoccupied cell $x$ of $B$, let $B + x$ be the board-state obtained by adding to $B$ a $P$-stone at $x$, and let $G + x$ be the

Figure 4. A white win-prelink ... and a corresponding win-link.
associated state, namely $G + x = [Q, B + x]$. Call a collection of sets mutually exclusive if the intersection of all the sets is empty. Van Rijswijck’s comments suggest the following rules for solving a state.

**Theorem 3** (Van Rijswijck, 2002) If $P$ (respectively $Q$) has a winning chain in $B$ then $P : \phi$ (respectively $Q : \phi$) is a win-link. If neither player has a winning chain in $B$, then

- $P$ wins $G$ if and only if $P$ wins $G + x$ for some move $x$; in this case, $G + x$ has some win-link $P : S$ and $G$ has a win-prelink $P : S + x$,

- $Q$ wins $G$ if and only if $Q$ wins $G + x$ for all moves $x$; in this case, for each $x$, $G + x$ has a win-prelink $Q : S_x$ and any collection $C$ consisting of one such $S_x$ for each $x$ is mutually exclusive; also, for each $C' \subset C$, if $C'$ is mutually exclusive then the union $U$ of the elements of $C'$ is a win-link $Q : U$ for $G$.

Figure 5 illustrates this theorem. The root state $G$ is a loss for White. Three of White’s possible moves are explored. In each state $G + x_i$, the move $y_i$ yields a black win; the resulting state $G + x_i + y_i$ has a black win-link $S_i$, so $G + x_i$ has a black win-prelink $S_i \cup y_i$; this win-prelink implies that $x_i$ loses in $G$, and moreover that any white move outside of $S_i \cup y_i$ loses. The set of these three win-prelinks is mutually exclusive. Indeed, the set containing just the win-prelinks $S_2 \cup y_2$ and $S_3 \cup y_3$ is already mutually exclusive, which means that the union of these two prelinks is a black win-link in $G$. It also means that the exploration of these two branches of the game-tree is sufficient to determine that White loses $G$; the consideration of any other move is unnecessary.

We omit the proof of correctness of the preceding theorem, which follows by elementary game-theory arguments from the fact that any Hex state has exactly one winner. Notice that these rules are by their definition complete: they can be used to solve any arbitrary Hex state.

From a computational point of view, the difficulty with both of these sets of rules is that the number of possible connection sets that can be computed in this way is exponential in the number of cells. For this reason, an exhaustive approach to computing connection sets based on either rule set will be forced to limit the number of intermediate connection sets computed. For example, Anshelevich’s (2002) game-playing program has maximum effectiveness when the number of $x$-to-$y$ connection sets stored is limited to about 40 per pair of cells $x, y$.

For both the static and dynamic computational processes, what is needed is some way of distinguishing those intermediate connection sets which are

---

3 This fact in turn requires some care to prove; see for example Beck (1969), Gale (1979), and Hayward and Van Rijswijck (200x).
Solving 7×7 Hex: Virtual Connections and Game-State Reduction

Figure 5. An example illustrating Theorem 3.

critical to solving the particular state from those which are not. We close this section by giving evidence that this is likely to be a difficult problem.

Assume that at some point in a computation involving the dynamic rules it is discovered that player $P$ has no winning move in a state $G$. It follows that $P$’s opponent $Q$ has a win-prelink $S_x$ after each possible move $x$ by $P$ and that the union of any collection of these win-prelinks which have an empty intersection establishes a win-link for $Q$. If $G$ is an intermediate state in the process of solving some earlier state, then $P$ needs to compute such a win-link to pass back to the state which gave rise to $G$. It is reasonable to expect that a useful win-link to pass back would be one that has the smallest number of cells, among all such possible win-links. However, it is also reasonable to expect this problem to be computationally difficult, since it seems to be intimately related to determining the outcome of a Hex game, which we have already noted is PSPACE-complete. Saks (2003) observed that this problem is indeed computationally difficult, as we now explain.

Formally, the Min-Union Empty Intersection Problem (MUEIP) is the decision problem which takes as input an integer $k$ together with a set $S = \{S_1, \ldots, S_t\}$ of subsets of a finite set $V$ and asks whether there is a subset $T$
of $S$ whose element-intersection is empty and whose element-union has size at most $k$. *Min Cover* is the decision problem which takes as input an integer $k$ together with a set $A = \{A_1, \ldots, A_t\}$ of subsets of a finite set $V$ and asks whether there is a subset of at most $k$ elements of $A$ whose union is $V$.

**Theorem 4** (Saks, 2003) MUEIP is NP-complete.

*Proof.* Consider an instance of Min Cover, where $k$, $A$, and $V$ are as defined above and $n = |V|$. This instance can be transformed in polynomial time into an instance of MUEIP, as follows.

For each index $j$, let $B_j$ be the set complement (with respect to $V$) of $A_j$, and let $B = \{B_1, \ldots, B_t\}$. Observe that the union of $k$ elements of $A$ is equal to $V$ if and only if the intersection of the corresponding $k$ elements of $B$ is empty. Let $V'$ be the set obtained by adding $t(n + 1)$ new elements to $V$. For each index $j$, let $B_j'$ be the set obtained by adding $n + 1$ of the new elements to $B_j$ in such a way that each $B_j$ gets expanded by a set of new elements disjoint from all other new elements. Let $B' = \{B_1', \ldots, B_t'\}$. Observe that a set of $k$ elements of $B$ has empty intersection if and only if the corresponding set of $k$ element of $B'$ has empty intersection, and this occurs if and only if the same set of $k$ elements of $B'$ has empty intersection and union with size at most $k(n + 1) + n$.

Since MUEIP is clearly in NP, the theorem follows from the preceding transformation and the fact that Min Cover is NP-complete (Karp, 1972). \qed

Since using virtual connections alone to solve arbitrary Hex states is likely to be computationally difficult, some extra game knowledge must be used to reduce the complexity of searching through the game-tree. We discuss some such reductions in the next section.

### 3. Move Domination and Game-State Reduction

One reason that Hex is a challenge for computers to play or solve is the high branching factor; especially in the early stages of the game, the number of possible moves is high. In this section we describe some move ordering information which considerably strengthens the algorithmic approach implicitly described by the virtual connection composition rules of the previous section.

A particularly useful form of move ordering information is move domination. Informally, one move dominates another if the former is at least as good as the latter. Since we are interested here only in solving states, namely in determining which player has a win-strategy, one move is “at least as good as” another if the former yields a win whenever the latter yields a win. Formally, for possible moves $u, v$ from a state $[P, B]$, we say that $u$ dominates $v$ if $P$ wins $[Q, B + u]$ whenever $P$ wins $[Q, B + v]$. 

Domination results are useful for our purposes since any dominated move can be ignored in searching for a winning move. Unfortunately, few results have been proved to date on domination in Hex. Beck (1969) proved that on an empty board size 2×2 or larger, moving to an acute corner (for example, A1 in Figure 7), is a losing, and so dominated, move. Using similar arguments, Hayward (2003b) recently obtained a move domination result involving certain three-cell configuration, as we now explain.

For a player $P$, a side cell is any cell which borders one of $P$’s two sides, a side pair $\{x_1, x_2\}$ consists of two adjacent side cells which border the same side, and a side triangle $(x_1, x_2, t)$ consists of a side pair $\{x_1, x_2\}$ together with a third cell, called the tip, adjacent to the two side cells. See Figure 7. A $P$-triangle is a side triangle belonging to $P$.

**Theorem 5** (Hayward, 2003b) Let $P$ be a player with opponent $Q$ and let $B$ be a board-state with an empty $P$-triangle $(x_1, x_2, t)$. For each subset $S$ of $T = \{x_1, x_2, t\}$, let $B + S$ be the board-state obtained from $B$ by adding a $P$-stone at each cell of $S$.

Then, for each $j = 1, 2$, $P$ wins $[Q, B + t]$ if $P$ wins $[Q, B + x_j]$. Also, $P$ wins any one of the four states $[Q, B + t]$, $[Q, B + \{t, x_1\}]$, $[Q, B + \{t, x_2\}]$, $[Q, B + \{t, x_1, x_2\}]$ if and only if $P$ wins all of them.

Our algorithm uses the above results in the following two ways. Firstly, for any state $[P, B]$ with an empty $P$-triangle, $P$ can ignore the two moves to the side of the triangle, since they are dominated by the move to the tip. Secondly, for any state $[Q, B]$ with a $P$-triangle with a $P$-stone at the tip and the two side cells empty, $P$-stones can be added to the two side cells, since this addition does not change the outcome of the game. As can be seen from Figure 12, the second result is particularly useful when combined with our virtual connection computation approach.

![Figure 6. Illustrating the second part of Theorem 5. Applying this result to the white side triangle with tip E2, it follows that a player has a winning strategy for one of these board-states if and only if that player has a winning strategy for all of these board-states.](image-url)
The Algorithm

Our algorithm SOLVER combines the approaches suggested by Theorems 1, 2, 3, and 5. For a player $P$ with opponent $Q$, the algorithm solves a state $G = [P, B]$ as follows.

**Algorithm SOLVER**($G = [P, B]$) For each side triangle for which the second part of Theorem 5 applies, add stones to the appropriate side cells; call the resulting board $B^*$. Statically compute virtual and weak connections. If a win-prelink for $P$ or a win-link for $Q$ is detected, then return the link (and if the win-link uses the tip of a triangle whose side was filled in, then add the side cells to the link).

Otherwise, let $T$ be the set consisting of all $Q$-win-prelinks for $G$ and let $R$ be the $P$-mustplay region. If $T$ is empty, then initialize $R$ to be all unoccupied cells; otherwise, initialize $R$ to be the intersection of all elements of $T$. Remove from $R$ any side-cells from any empty $P$-triangle. While $R$ is not empty, pick a cell $x$ in $R$, and do the following:

Let $B^*_x$ be the state obtained from $B^*$ by adding a $P$-stone at $x$ and, if $x$ was the tip of an empty $P$-triangle before this move, filling in the triangle. Recursively solve $G_x = [Q, B^*_x]$.

If $P$ wins $G_x$, say with win-link $X$, then add to $X$ the cell $x$ as well as the two associated side-cells if $x$ was the tip of an empty $P$-triangle, and exit the while loop and return. If $Q$ wins $G_x$, say with win-prelink $X$, then add $X$ to $T$.

If the while loop terminates without discovering a win-prelink for $P$, then the union of elements of $T$ forms a win-link for $Q$.

A sample execution of the algorithm is described in Figures 7 through 9. The correctness of our algorithm follows easily from the previous theorems; we omit the proof.

![Figure 7](image)

**Figure 7.** SOLVER solves b6: initialization. After the initial move (left), the game-state is reduced by applying Theorem 5 and adding white stones to the two side-cells of the white side-triangle with tip b6. In the resulting state, White has two win-prelinks (center-left and center-right) whose resulting intersection yields a 13-cell black mustplay region (right). If Black has a winning move, it has a winning move to one of these 13 cells.
Solving 7×7 Hex: Virtual Connections and Game-State Reduction

Figure 8. SoLVER solves b6-c4. As shown by the SoLVER b6 recursion tree in Figure 13, c4 is the first black response considered to the white b6 opening (left). Following the topmost path b6-c4-f2-d5-d4-c5-e5-e4-g3-f3-g2-f4 in the recursion tree and applying Theorem 5 after f2 leads to the first solved state (center, with white win-prelink); since f4 is a leaf of the recursion tree, the white win-prelink here was discovered statically. SoLVER continues solving the c4-subtree, eventually determining that c4 is a black loss (right, with white win-prelink). This win-prelink does not contain c4 or b5, so, of the 13 possible b6-responses corresponding to the initial black mustplay region described in Figure 7, 11 moves remain to be checked.

Figure 9. SoLVER solves b6: conclusion. The move to f1 is the last black reply considered in response to the white b6 opening (left, with white win-prelink), since after the discovery of this last white win-prelink, the set of such win-prelinks has empty intersection. The union of these 11 white win-prelinks gives the final win-link for White (right).

5. SoLVER 7×7 1-Opening Solutions

As mentioned earlier, SoLVER is strong enough to solve arbitrary 7×7 states. Figures 10 and 11 summarize the results obtained by running SoLVER on all 49 7×7 1-openings. Figures 13 and 14 show the SoLVER recursion trees from two of these executions, while Figure 15 shows a longest line of play from each of the 49 solutions. Each execution was performed on a single processor machine; in each case, the run time was roughly proportional to the number of nodes in the SoLVER recursion tree, taking about one minute for the five 1-openings with the smallest node-counts, and about 110 hours for the 1-opening with the largest node-count; the total run time for all 49 1-openings was about 615 hours. A listing of all 49 trees (including a tree viewer) is available at http://www.cs.ualberta.ca/~hayward/hex7trees.

5The program was compiled with gcc 3.1.1 and run on an AMD Athlon 1800+ MHz processor with 512 MB memory running Slackware Linux.
Figure 10. All 7×7 1-opening results, as found by SOLVER. The stone on each cell indicates the winner with perfect play if White’s first move is to that cell. The move indicated on each losing cell is the winning countermove discovered.

Figure 11. Number of nodes in the SOLVER 7×7 1-opening recursion trees.

For any size Hex board, the set of winning open-move cell locations is symmetric with respect to reflection through the center of the board. Notice that the SOLVER node-counts do not share this symmetry, as neither the order in which SOLVER considers moves nor the static computation of virtual connections is designed to reflect this symmetry.

Figure 12 demonstrates the relative strength of the three key parts of our algorithm, namely virtual connection computation, side-triangle move domination, and side-triangle fill-in, by showing SOLVER node-counts when various of these features are turned off. In particular, notice that adding side-triangle fill-in to virtual connection computation results in a substantial decrease in the number of nodes considered, while further adding side-triangle domination has little effect.
Solving $7 \times 7$ Hex: Virtual Connections and Game-State Reduction

In comparing the winning $7 \times 7$ opening moves (Figure 10) with winning opening moves on smaller boards (Figure 2), some features common to each of these $n \times n$ boards are worth noting. For example,

- the $n$ cells on the short diagonal (obtuse corner to obtuse corner) are all first-player winning openings,

- the $n - 1$ cells on each of the first-player’s sides (except for the cell in the short diagonal) are all first-player losing openings.

It would be of considerable interest to show whether these results hold in general, especially if the proof is positive (as opposed to say a single counterexample), since to date, for arbitrarily large $n \times n$ boards,

- no particular move is known to be a first-player win,

- the only moves which are known to be first-player losses are
  - for $n \geq 2$, the two acute corner cells (Beck, 1969)
  - for $n \geq 3$, the two cells each in the first-player’s side and adjacent to the acute corner cell (Beck, 2000).
Figure 13. The SOLVER recursion tree for the 7×7 opening White-b6 (with the ten nodes connected by dotted edges added so that every path ends with a winning move). For each node, the order of child generation is top-to-bottom. Each SOLVER recursion tree is a subtree of the complete game-tree, as the only replies to a winning move which appear in the recursion tree are those replies in that state’s mustplay region. For example, consider for the tree shown here the state G after White plays b6. As shown in the second diagram in Figure 7, White has a winprelink created by playing at c4 which does not contain d4; thus d4 is not in the black mustplay region for G, so SOLVER never needs to consider the black move to c4, so c4 does not appear as a child of b6 in this recursion tree. Notice from the tree shown here that in solving the b6 opening the selection of d2 as the first move considered at the b6-c5-c3-c2 subtree was unfortunate, as d2 leads to a white loss whereas f2, the second move considered, leads to a white win. If f2 had been considered first, the d2 subtree would not have been explored, and the resulting recursion tree would have had only 97 nodes instead of 197.
Solving 7×7 Hex: Virtual Connections and Game-State Reduction

Figure 14. The SOLVER recursion tree for the 7×7 opening White-f1 (with the five nodes connected by dotted edges added so that every path ends with a winning move). For each node, the order of child generation is top-to-bottom. Notice that the f1-b6 subtree, which establishes that b6 is a winning countermove to f1, is paradoxically smaller than the b6 subtree shown in Figure 13, in part because of the move ordering here is more fortunate than there. In this f1-tree, whenever it is White’s turn to play, the first move considered turns out to be a winning move; this is not the case in the b6 tree shown in Figure 13.
Figure 15. Longest 7x7 SOLVER lines of play. For each of the 49 7x7 1-openings, the corresponding line shows a longest line of play from the the associated SOLVER solution. The top row shows the move number of that column.
6. Conclusions and Open Problems

We have shown how combining static and dynamic virtual connection computation methods with some move domination results yields an algorithm strong enough to solve arbitrary $7 \times 7$ Hex states. A next step is to design an algorithm strong enough to solve $8 \times 8$ states; preliminary results suggest that this is considerably more difficult and that further techniques will be required. Another direction is to use SOLVER to gather $7 \times 7$ information which can be used to find better move ordering heuristics for Hex game-tree search on (much) larger boards; for example, such data would be useful in analyzing any local configuration with effective board size at most $7 \times 7$.

Acknowledgements

The authors gratefully acknowledge the support of the Natural Sciences and Engineering Research Council of Canada, the University of Alberta Research Excellence Envelope, and the University of Alberta GAMES Research Group. Also, the fourth and fifth authors gratefully acknowledge the support of an NSERC Summer Undergraduate Research Award.

We thank Michael Buro, Maryia Kazekevich, Martin Müller, and Jonathan Schaeffer for their assistance in sustaining the Mongoose Hex project which was the starting point for this work. We also thank the referees for their detailed comments on an earlier version of this article.

References


AUTOMATED IDENTIFICATION OF PATTERNS IN EVALUATION FUNCTIONS

T. Kaneko, K. Yamaguchi, S. Kawai

Graduate School of Arts and Sciences (Kaneko, Kawai) and Information Technology Center (Yamaguchi), The University of Tokyo, Tokyo, Japan

{kaneko,yamaguch,kawai}@graco.c.u-tokyo.ac.jp, http://www.c.u-tokyo.ac.jp/kaneko/

Abstract

This paper proposes a general and automated method that generates accurate evaluation functions, without expert players' knowledge of a target game. Patterns (which are partial descriptions of a game state) are widely used as primitives of evaluation functions in game programming. They have to be carefully selected in order to generate accurate evaluation functions. Our approach consists of three steps: (1) generation of logic formulae by using the specifications of a target game, (2) translation of the formulae into patterns, and (3) selection of a set of suitable patterns from those generated. The problem, in the automated identification of suitable patterns, is that it is difficult either to generate only useful patterns or to examine all possible patterns. The latter obstacle is due to the prohibitive numbers involved. We solved this dilemma by a combination of two methods, where one method generates patterns of good quality, and the other method entails a lightweight selection based on statistics that could handle a large number of candidates. Experiments in Othello revealed that about 100,000 patterns from more than eight million automatically generated patterns could be successfully selected with our method, and that accurate evaluation functions were constructed. This accuracy is comparable to that of specialized Othello programs and is much better than that of the evaluation functions generated by existing general methods.

Keywords: feature generation, feature selection, evaluation function, Othello

1. General Game Players

One of the most ambitious goals of artificial-intelligence research is the development of a general game player that can learn and play an arbitrary instance of a certain class of game. Strong game programs must have an accurate and efficient evaluation function that can estimate the results of a game based on the notion position. Since an evaluation function is specific to a target game, the development of general game players requires evaluation functions to be automatically constructed without assistance of human experts.
1.1 Learning of Evaluation Functions

A popular way of constructing an evaluation function is to make it a (linear) combination of evaluation primitives called features, and adjust the parameters of the combination (Samuel, 1967; Tesauro, 1992; Buro, 2002). Generally, the construction of evaluation functions requires the acquisition of features, and the training of a prediction model (e.g., linear combination).

1.2 Learning of Features

The main difficulty in constructing evaluation functions is identifying appropriate features. In most preceding investigations, these features have been provided by human experts for the game involved. Our first goal is to identify appropriate features mechanically. To achieve this we employed a method of constructing features written in logic programs (we called them logical features). However, logical features are not practical because they are too slow in evaluating logic programs. Yet, the advantage is that practical evaluation functions were constructed with a large number of patterns as features (Buro, 1998; Buro, 2002). A pattern is a logical formula in a specific form. We introduce a rigorous definition for this in Subsection 3.3. Even though a pattern is just a logical formula in a specific form, the mechanical identification of suitable pattern sets to derive a good evaluation function is a difficult task.

1.3 The Approach

Our second goal is to construct efficient and accurate evaluation functions through game-independent methods. Here we propose a combination of methods that yields patterns similar to Buro’s (1998) methods by translation from logical features. These methods are:

1. generation of logical features,
2. extraction of patterns from logical features, and
3. selection of suitable patterns.

A large number of patterns are produced in steps 1 and 2, and useful patterns are selected in step 3. The claim of the paper is that this selection is indispensable for generating useful evaluation functions. The reason why we have to generate such a large number of patterns in steps 1 and 2 is that they are required to achieve accuracy in the evaluation functions constructed. There is no known method of generating only useful patterns.

The method of selection must be so lightweight that a machine can evaluate numerous pattern candidates within practical time limitations. We demonstrate the effectiveness of our solution through experiments.
The paper is organized as follows. Section 2 reviews related work and other issues that need to be resolved to construct general game players. Section 3 introduces the basic terminology. Methods to generate logical features and evaluate positions are briefly explained in Sections 4 and 5. In Section 6 a method of selection is proposed. Section 7 shows the experimental results in Othello. Section 8 concludes the paper.

2. Related Work

The construction of general game players requires the acquisition of game-specific search enhancements as well as evaluation functions, such as realization probabilities (Tsuruoka, Yokoyama, and Chikayama, 2002), opening books (Lincke, 2001), and endgame books. This paper only addresses evaluation functions, even though we are aware that our method can be applied to the acquisition of other knowledge.

In constructing evaluation functions, the training of prediction models requires unbiased training positions and an appropriate labeling (Buro, 1998). It is well known that the usefulness of learned evaluation functions depends on the training positions used. Thus, unbiased positions are needed to develop strong programs. Because this paper primarily focuses on the acquisition of features, the experiments were conducted on a game where both the training positions and the labeling were available (near endgame in Othello). In games where these are not available, we can apply methods of gathering positions via self-play and temporal-difference learning (Tesauro, 1992, 2002).

We simply use linear regression for prediction because we could use a method that iteratively adjusts the weights in a linear model, even when a very large number of features are used (Barrett et al., 1994). Other prediction models, such as neural networks, could be used with our method, too.

Logical features are general and were actually applied to many games. We mention Othello and a single-agent search problem by Fawcett (1993), symmetric chess-like games (Pell, 1993) and a variant of Shogi (Kaneko, Yamaguchi, and Kawai, 2002). However, the cost of evaluating positions is prohibitive when there are logical features due to the slow evaluation of logic programs, despite the recent efforts that have increased speeds more than 4,000 times (Kaneko, Yamaguchi, and Kawai, 2000, 2001).

Buro (2002) used patterns in fixed shapes. This is effective in achieving highly efficient pattern matching, even when a large number of patterns is involved. However, there is no established method of identifying effective shapes mechanically, and we do not know whether patterns in such fixed shapes are useful in other games. Kojima, Ueda, and Nagano’s (1997) method acquired patterns from game records in Go through genetic programming. This requires
game-specific adaptation to apply it to other games because it depends on the importance of adjacent stones.

We recently developed a method of generating patterns from logical features (Kaneko et al., 2001). However, the accuracies of the generated evaluation functions did not reach those that Buro obtained. This is because we only used about 4,000 patterns, while Buro used about 200,000. For our method it was impossible to provide a sufficient number of useful patterns because effective methods of selection were up to then unknown.

The selection of features is a central research topic in artificial intelligence, and many methods have been developed (Guyon and Elisseeff, 2003; Jain, Duin, and Mao, 2000). It is a combinatorial optimization problem. Heuristics are essential because the computational costs identifying an optimal pattern subset are known to be exponential in terms of the number of candidates (Jain, Duin, and Mao, 2000). Such costs are not acceptable. Moreover, popular selection methods such as the F-test in statistics cannot be used here. To illustrate this difficulty, we used about eight million candidates in the experiments that will be described later. Obviously, their covariances cannot be stored on normal computers.

3. Basic Terminology

This section introduces the basic terminology, including the specifications of a game written in logic (Subsection 3.1), the logic features (Subsection 3.2) and the definition of patterns (Subsection 3.3).

3.1 Positions and Domain Theory

A position is an intermediate status of a game. It is described by a set of special facts. A fact is a clause without a body. In Othello, owns and blank
Automated Identification of Patterns in Evaluation Functions

legal_move(S, Player):~square(S), bs(S, _End, Player).
bs(S1, S3, P):~blank(S1), opponent(P, Opp),
neighbour(S1, D, S2), span(S2, S3, D, Opp),
neighbour(S3, D, S4), owns(P, S4).
span(S1, S2, D, Owner):
    square(S1), square(S2), player(Owner), owns(Owner, S1),
    neighbour(S1, D, S3), span(S3, S2, D, Owner).
span(S, S, D, Owner):
    square(S), player(Owner), owns(Owner, S), direction(D).
line(S, S, D):~square(S), direction(D).
line(From, To, D):~neighbour(From, D, Next), line(Next, To, D).
opponent(x, o). opponent(o, x).
direction(n). direction(ne). direction(e). direction(se).
direction(s). direction(sw). direction(w). direction(nw).
square(a1). square(a2). square(a3). (⋯)
square(d2). square(d3). square(d4). (⋯)
neighbour(a1, s, a2). neighbour(a2, n, a1).
neighbour(a2, s, a3). neighbour(a3, n, a2). (⋯)
neighbour(c4, ne, d3). neighbour(d3, sw, c4). (⋯)

Figure 2. Sample domain theory for Othello.

are used to represent a position. To demonstrate this, we have shown the facts defined in the initial position in Othello and the position after Black has played c4 in Figure 1. Here, Black is denoted by x, and White is denoted by o. In the initial position, owns(d5, x), owns(e4, x), owns(d4, o), and owns(e5, o) are defined for squares with a disc, and blank is defined for each empty square.

The main part of the specifications of a game consists of the rules of the game and the goal conditions. This is called domain theory and described by a set of Horn Clauses. The example Othello domain theory in Figure 2 is used throughout this paper.

3.2 Logical Features

Logical features are defined as Horn Clauses of the predicate logic where predicates in their body are defined by domain theory or position. The following clause is an example of a logical feature.\(^1\)

\[ \text{f(A)}: \text{owns}(x, A) \]  
% pieces for Black

\(^1\)This is written as "\(f(N) : \text{~count}([A], \text{owns}(x, A), N)\)" in Fawcett (1993). In this paper, "count" has been assumed to be the default semantics of logical features and has therefore been omitted.
The *value* of a logical feature for a state is defined as the number of solutions, where solutions are the bindings of such constants to variables that make the clause true. In the above feature, $A$ is a variable, and the solutions in the initial position in Figure 1 (left) are $d5$ and $e4$ (two solutions), which is the number of squares currently owned by Black.

### 3.3 Patterns

A pattern is defined as a conjunction of facts describing a part of a position. The value of a pattern is 0 or 1 according to its Boolean value; in a given position this value of a fact is 1 if it is defined (or 0 if undefined). For example, the following is a pattern.

$$\text{blank}(a1) \land \text{owns}(x,a2) \land \text{owns}(o,a3)$$

This pattern is a logical formula for “White can play on square a1.”

### 4. Pattern Generation

Patterns are generated through the following steps:

1. generation of logical features with Fawcett’s (1993) method,
2. translation of logical features into propositional logic by unfolding, and
3. extraction of patterns from propositional logic.

First, logical features are generated by means of syntactic translation of Horn Clauses, which are extracted from the domain theory of a target game. For example, the following feature (called a mobility feature) can be generated.

$$f(A): \neg \text{legal.move}(A,o). % \text{mobility for White}$$

Complex features can be generated by taking the preconditions of existing features. Fawcett (1993) has more details on automated construction.

In the next step, generated features are translated into propositional logic by *unfolding*. This is a technique in partial evaluation of logic programming (Bossi, Cocco, and Dullie, 1990), and is repeatedly applied until features only consist of ground facts. In conventional games with reasonable rules, it is easy to write a domain theory so that the unfolding of generated features stops even if they contain recursively defined clauses, due to the finiteness of the number of squares and satisfiable terms. Detailed translation methods have been described by Kaneko et al. (2001). The following clauses are part of the results we obtained for the unfolding of the feature in the above example.

$$\text{legal.move}(a1,o) :\neg \text{blank}(a1), \text{owns}(x,a2), \text{owns}(o,a3).$$
$$\text{legal.move}(a1,o) :\neg \text{blank}(a1), \text{owns}(x,b1), \text{owns}(o,c1).$$
$$\text{legal.move}(a1,o) :\neg \text{blank}(a1), \text{owns}(x,b2), \text{owns}(o,c3).$$
Finally, we extracted patterns from the unfolded features simply by taking the conjunctive part of their propositional formulae. The following formulae are patterns extracted from the unfolded features listed above.

- $\text{blank}(a_1) \land \text{owns}(x,a_2) \land \text{owns}(o,a_3)$
- $\text{blank}(a_1) \land \text{owns}(x,b_1) \land \text{owns}(o,c_1)$
- $\text{blank}(a_1) \land \text{owns}(x,b_2) \land \text{owns}(o,c_3)$

Each pattern has a corresponding clause whose body (right hand of clause) is equivalent to the pattern.

5. Pattern Matching

Below, we briefly discuss a pattern matching method to justify the selection method of the next section. The purpose of the selection is to identify sets of patterns that produce efficient and accurate evaluation functions, where their efficiency depends on how the patterns are evaluated. Basic ideas in efficient matching are (1) performing incremental calculations and (2) utilizing a partial order on patterns.

5.1 Incremental Matching with a Diagram

Incremental matching was efficiently implemented with a Hasse diagram (Gries and Schneider, 1993) on the partial order of patterns, as outlined in Figure 3. Let each $a$, $b$, and $c$ be a fact describing a position (such as $\text{blank}(a_1)$), and consider that there are six patterns $\{abc, ab, bc, a, b, c\}$. Here, $ab$ means the conjunction of $a$ and $b$. In the figure, a pattern is denoted by a square, and a fact is denoted by a circle. For each pattern, the question whether matching is required can quickly be determined by using the diagram. For example, matching of pattern ‘$abc$’ is only required when the value of pattern ‘$ab$’ or ‘$bc$’ changes.

![Figure 3. Hasse diagram of sample patterns.](image)

The computational costs of incremental matching can be estimated by the number of nodes visited. Because each edge will be visited once at most, the cost for the worst case is proportional to the number of edges. Cube extraction (Rudell, 1996) was applied to a diagram here to reduce edges, as well as other optimizations. Details are discussed in Kaneko et al. (2001).
5.2 Counters for Matching

To speed up matching of individual patterns, an integer counter $cur(p)$ was associated with each pattern $p$ such that the matching was determined by integer comparison instead of naively computing the logical conjunction of each fact in the pattern.

Let $dep(p)$ ($upd(p)$) be children (parents) of pattern $p$ in a diagram. Counter $cur(p)$ is defined as the number of children of $p$ whose current value is true. Then, as long as $cur(p)$ is properly maintained, the Boolean value of $cur(p) = |dep(p)|$ coincides with the value of pattern $p$.

6. Pattern Selection

This section introduces a lightweight selection method, which consists of two methods that are computationally inexpensive. These are:

- *preliminary filtering* by using the frequency of patterns, and
- *approximated forward selection* by assessing the contribution of patterns to the accuracy of a prediction model.

The latter method takes into account the accuracy of a linear model that uses selected patterns. Consequently, it requires that the model is trained (by weight fitting) for each subset of patterns; thus the method is relatively expensive. The former method is more efficient because it only uses the frequency of each pattern. However, it cannot be used to select useful patterns by itself. Hence, we first need to filter the candidates with the former method, and then select useful patterns with the latter method, to reduce its weight-fitting time.

6.1 Preliminary Filtering by Frequency

First, useless patterns are heuristically determined and filtered out by analysing their frequency, before approximated forward selection is done in the next step. There are two background considerations: (1) if low-frequency patterns are used, the efficiency of evaluating positions by using the method detailed in Section 5 will improve, and (2) the use of extremely low-frequency patterns tends to cause over-fitting. We claim that high- or extremely low-frequency patterns can safely be rejected without a loss of quality in the generated evaluation functions. Although this may seem similar to the filtering in existing work (e.g.,

---

2 More precisely, it is better to measure the frequency at which the patterns change from one position to another to improve search efficiency. We used the frequency of the patterns themselves, because this could be measured more easily. Moreover, to reduce the computational costs of weight fitting, reducing the frequency of the patterns themselves is also essential, as discussed in Subsection 7.2.2.
Automated Identification of Patterns in Evaluation Functions

Kojima et al., 1997; Buro, 1998), our approach is different in the sense that our colleagues do not explicitly reject high-frequency patterns as we do.3

We measured the frequency of part of Buro’s (2002) horizontal patterns to estimate an appropriate frequency range (Figure 4). Because the highest frequency in Buro’s patterns was 0.075, we expected that good evaluation functions could be generated with only patterns with a frequency below this value. Figure 4 also has the results of measurement for our patterns. It can be seen that many patterns can be filtered out by frequency. The preferable frequency ranges were determined by the experiments, which are discussed in Section 7.

6.2 Approximated Forward Selection

After filtering, we applied a method of statistically selecting explanatory variables, by treating a pattern as a binary variable. Approximated sequential forward selection was adopted from many existing methods (Guyon and Elisseeff, 2003). It is so efficient that it was used already for manual computation before computers became widely available (Okuno et al., 1981).

The algorithm is listed in Figure 5. It is used to select a subset of variables (S) that are effective in predicting a target variable (y0), from a set of candidates (i.e., patterns, X). A target variable is the difference between the number of black and white discs (explained below in the experiments).

One pattern (xαi) is added to the selected set (S) at the seventh line for each loop, as in sequential forward selection. A priority function, also explained later, is used to select a pattern. Let n be the number of candidates and m be the number of patterns finally selected. Because variables in S are never removed, the method tries m subsets of candidates, which is far less than the possible number of subsets, 2n.

3Kojima et al.’s (1997) method and the inductive algorithm proposed by Buro (1998), which was not used in preparing the evaluation functions for LOGISTELLO, tend to discard high-frequency patterns because they prefer specific patterns in matching. As patterns for specific given shapes contained at least eight squares, high-frequency patterns were not used in constructing evaluation functions for LOGISTELLO.
\[
\text{\begin{verbatim}
\text{\begin{verbatim}
\end{verbatim}}
\end{verbatim}}
\]

Figure 5. Approximated forward selection algorithm.

A priority function is used to estimate the usefulness of the pattern for selection at the seventh line, and this should be carefully adopted taking the purpose of selection into consideration. For practical game programming, efficiency and accuracy should be taken into account to estimate the usefulness of a pattern in terms of priority. In this paper, we used the correlation with residuals after the \(i\)-th regression \(y_i\) as a priority function to achieve accuracy.

Here, if explanatory variables have no correlation with one another, variables selected with this method are equivalent to the ones selected by normal sequential forward selection, where the multiple regression coefficient in predicting \(y_0\) using all variables in \(S\) is used as the priority function.\(^4\) The order of candidates affects the results in other cases (Okuno et al., 1981). However, this method is more efficient than sequential forward selection because it uses univariate regression instead of multivariate regression.

\[
\text{\begin{verbatim}
\text{\begin{verbatim}
\end{verbatim}}
\end{verbatim}}
\]

Figure 6. Iterative selection algorithm.

The improved computation applied so far leads to appropriate results. Yet, the most expensive computation is to determine the priority (i.e., correlation) of each pattern in each loop. Naively, it requires pattern matching over all training positions for every loop, but then the computational costs are unacceptable. The

\(\text{\begin{verbatim}
\text{\begin{verbatim}
\end{verbatim}}
\end{verbatim}}\)

\(^4\)The method approximates sequential forward selection by using the accumulation of univariate regressions instead of multivariate regression.
priority of each pattern can be incrementally updated by means of a table holding the number of pattern co-occurrences if there are not too many candidates.

Thus, to avoid frequent pattern matching, patterns were split into sets of a moderate number of patterns \( \{X_0, X_1, \ldots, X_n\} \) in advance, and approximated forward selection was iteratively applied to each \( X_i \) in turn, as shown in Figure 6. A test to determine whether the priority of a selected pattern went beyond a given threshold worked well as a termination criterion in each approximated forward selection. We selected variables from \( X_0 \) up to a given threshold, and then selected variables from \( X_1 \) up to the given threshold. This step was repeated to \( X_n \). Preferable priority thresholds were estimated in the experiments and are discussed in Section 7. There were 1,000 candidates \( (X_i) \) in each approximated forward selection in our experiments. Although accuracy improves with greater numbers, only slight improvements could be observed for 4,000 candidates in our experiments.

7. Experimental Results

We did experiments on Othello to prove the effectiveness of the generation and selection methods proposed. We compared evaluation functions generated by our methods with those generated by other general methods, and with the evaluation functions used in specialized Othello programs. We used a computer with an Athlon MP 2100+ CPU (1.7 GHz) for these experiments. The program was implemented in GNU C++.

7.1 Pattern Generation and Selection

First, 11,079 logical features were generated by Fawcett’s (1993) method. Subsequently, 8,502,664 unique patterns were extracted from the logical features with the method proposed in Section 4. We then did selection by frequency as described in Subsection 6.1. Several sets of patterns were selected with various frequency ranges. Finally, we applied the iterative selection described in Subsection 6.2 to the resulting sets with various priority thresholds. The priority function used here was correlation, and candidates were sorted by frequency.

7.2 Accuracy of Evaluation Functions

This subsection contains the heart of our experimental research. It is subdivided into six sub-subsections, each of them dealing with a relevant item.

7.2.1 Training Positions and Labeling. Evaluation functions made up of selected patterns were constructed to enable the usefulness of patterns to be estimated. Each of the functions was a linear model of patterns. The weights were adjusted by means of least mean squares to predict the final score (difference between number of black and white discs at the end of the game...
after both players had played the best moves). We separately constructed the evaluation functions for the positions of 60 discs and those for 55 discs. We only used the positions of 60 and 55 discs because positions of the near the endgame can be immediately labeled with the results of a complete search.

The positions we used in selection and training were extracted from games played between LOGISTELLO and KITTY.\textsuperscript{5} It should be noted that our proposed method works without the game records of strong game programs. The purpose of using positions taken directly from games is to gather unbiased positions and to demonstrate the method's learning ability in positions that strong programs face. About 50,000 positions were selected by eliminating duplicate positions considering the symmetry of the geometry and players. We then generated two disjoint sets of positions expanding the symmetric ones.\textsuperscript{6} One set contained about 800,000 positions for training and the other had about 6,000 positions for testing.

7.2.2 Adjustment of Weights. Weights in evaluation functions with fewer than 10,000 patterns were adjusted with LAPACK\textsuperscript{7}, and an iterative method (BiCGSTAB\textsuperscript{8}) (Barrett et al., 1994)) was used instead, due to memory limitations, in other cases. The time for weight fitting depends on the efficiency of an evaluation function and on the number of matching patterns for a position on average. This efficiency was primarily important because the iterative method requires pattern matching over all training positions for many repetitions. The number of multiplications required for each position is about the number of matching patterns squared. Thus, it was not feasible to use all patterns generated without selection. The time for weight fitting tended to be more than a week if there were more than 100,000 patterns. Buro could use more patterns because his efficiency is much better, as will be described below, and because only 50 patterns at most should match each position due to the carefully crafted shapes.

7.2.3 Accuracy of Proposed Evaluation Functions. The graph in Figure 7 illustrates the accuracy of our evaluation functions and the others. Here, "error" in the vertical axis is the square root of mean square errors. The horizontal axis plots the number of patterns on a logarithmic scale. Our evaluation functions ("with selection") for positions with 60 discs are denoted by the '+', and those for positions with 55 discs are denoted by the '■'. The errors for 55

\textsuperscript{5}Both are available at ftp://external.nj.nec.com/pub/igord/IOS/misc/.
\textsuperscript{6}To generate evaluation functions that yield the same value at symmetric positions, symmetric patterns should have the same weight in the evaluation functions. We achieved this by simply instantiating all symmetric positions when adjusting weights.
\textsuperscript{7}http://www.netlib.org/lapack/
\textsuperscript{8}http://netlib2.cs.utk.edu/linalg/html_templates/Templates.html
discs are larger than those for 60 discs, because it is more difficult to predict scores for the positions of earlier game stages. The frequency ranges used in filtering were \([1.25 \cdot 10^{-5}, 0.075]\); 540,724 patterns were selected. The priority thresholds used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01; we selected approximately 3,000 to 140,000 patterns. The accuracy of our evaluation functions improved as the number of patterns increased.

7.2.4 Comparison with Logical Features or All Patterns. We compared our evaluation functions with those using logical features and those using all patterns without selection to demonstrate improvements over existing general methods.

We have already reported on a comparison of all patterns and logical features (Kaneko et al., 2001). The accuracy of evaluation functions using 18 logical features that Fawcett had selected was 12.9 and 12.5, and the accuracy of evaluation functions with 42 logical features that were statistically significant and selected with an F-test from 10,000 features was 8.90 and 12.4 for positions with 60 discs and 55 discs, respectively. Evaluation functions with logical features were more than 20 times slower than those with patterns extracted from the same logical features. The results indicate that extracted patterns are much more effective than the logical features themselves.

In Figure 7, “without selection” means the accuracy of evaluation functions that use automatically generated patterns without selection. The accuracy of our evaluation functions was far better than that of patterns without selection (plotted with ‘*’ and ‘•’). The accuracy of the latter functions were established in and taken from the authors’ previous work (Kaneko et al., 2001). The results indicate that the proposed methods are more effective than existing general methods.

![Figure 7. Accuracy of evaluation functions.](image-url)
7.2.5 Comparison with Buro’s Patterns. We compared our evaluation functions with those of a specialized Othello program to evaluate our accuracy. In Figure 7, “Buro” means our previous reproduction of Buro’s method (Kaneko et al., 2001). The accuracy of our evaluation functions improves as the number of patterns increases, going beyond that of Buro’s (plotted with the ‘○’ and ‘×’). The results indicate that accurate evaluation functions are mechanically generated, without having to incorporate manually important shapes in Othello.

7.2.6 Comparison with Randomly Generated or Selected Patterns. To demonstrate the importance of both pattern generation and selection, we constructed evaluation functions with random generation/selection instead of the proposed generation/selection, and compared their accuracies.

Random Generation + Proposed Selection. In Figure 7, “random + selection” means evaluation functions that use patterns selected with our method, from randomly generated patterns instead of the ones generated by this method. First, 8,502,664 patterns were generated, each of which was a conjunction of the randomly selected status of squares. Then, about 3,000 and 6,000 patterns were selected with the selection we propose. The difference between the accuracy of randomly generated patterns and that of ours means that our method of generating patterns is indispensable in producing useful patterns.

Proposed Generation + Random selection. We measured the accuracy of evaluation functions with 4,147 patterns that were randomly selected instead of with the selection we propose. The error was more than 14.5 and is not plotted in the graph. The difference between the accuracy of randomly selected patterns and that of our method means that our pattern selection method is indispensable in producing useful patterns.

7.3 Efficiency of Evaluation Functions

Figure 8 illustrates the efficiency and accuracy of our evaluation functions selected for various frequency ranges. The horizontal axis plots the number of patterns used in the evaluation functions and the vertical axis plots efficiency by the number of positions evaluated in one second. The priority thresholds we used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01. As the number of patterns increased, the efficiency of evaluation functions deteriorated while the accuracy improved, almost regardless of frequency ranges. For this experiment, we collected a sequence of about 3,000,000 positions. Then the df-pn·
Automated Identification of Patterns in Evaluation Functions

Figure 8. Efficiency of evaluation functions for various numbers of patterns (55 and 60 discs).

Figure 9. Accuracy of evaluation functions with various priority thresholds (55 and 60 discs).
search (Nagai and Imai, 1999) visited the root positions of 49 discs, which were extracted from 23 matches in IOS records.9

Although the efficiency of our evaluation functions was much better than the efficiency of evaluation by logical features (Kaneko et al., 2001), it was worse than that of a specialized Othello program. LOGISTELLO's speed was about 270,000 nodes/sec when running on a Pentium-II 333 MHz (Buro, 1998). This speed would have been about 1.4 million nodes/sec (by extrapolation) if it had been run on a 1.7-GHz CPU. Further research is required to make practical evaluation functions because efficiency is usually more important than accuracy.10 These differences were partly because we did not take efficiency into account in the selection of patterns and partly because we could have used a much more efficient pattern matching algorithm than the one we proposed if we had restricted our patterns to Buro's (1998) shapes.

7.4 Parameters for Selection

To determine appropriate values for frequency ranges and priority thresholds so that the proposed selection would work well, we investigated their influence on the efficiency and accuracy of the generated evaluation functions and on the time required for selection.

The graphs in Figure 9 plot the accuracy of our evaluation functions for positions with 60 and 55 discs, consisting of patterns selected with various frequency ranges and various priority thresholds. We can see that the frequency ranges do not distinctly affect the quality of selected patterns, if its upper boundary is greater than 0.15. Thus, we concluded that the accuracy of evaluation functions is mainly determined by the number of patterns used in them.

The priority thresholds used in selection determine the number of patterns that are finally selected. Figure 10 plots the relation between the number of

---

9These are available at ftp://external.nj.nec.com/pub/igord/othello/ios/.
10Future advances in hardware will favour the accuracy because these will eventually compensate for serious delays when in-depth searches reach a saturation point (Heinz, 2001).
selected patterns and priority thresholds. The vertical axis plots the number of patterns on a logarithmic scale, and the horizontal axis plots the priority thresholds. Here, we used correlation for priority. We can see that the number of patterns selected is mainly determined by priority thresholds regardless of frequency ranges (denoted by symbols), and that the symbols in the graph are plotted at almost the same location if the same priority thresholds are used. Also, larger numbers of patterns are selected as lower thresholds are used. Thus, one can control the trade-off between the accuracy and efficiency of evaluation functions by adjusting the priority thresholds, because these are mainly determined by the number of patterns in them as previously discussed.

The time for iterative selection depends on frequency ranges as well as the number of selected patterns. Figure 11 plots the relation between time and the number of selected patterns with various priority thresholds and frequency ranges. The priority thresholds we used were 0.000125, 0.0005, 0.00125, 0.005, and 0.01. The horizontal axis plots the number of patterns finally selected by iterative selection, and the vertical axis plots the time for selection in minutes. These results are acceptable because we have to inspect a larger number of candidates during iterative selection for frequency ranges with larger upper bounds.

Figure 12 plots the relation between the efficiency and accuracy of evaluation functions. The vertical axis plots accuracy by the square root of mean square errors, and the horizontal axis plots efficiency by the number of positions evaluated in one second. The one right below is to be preferred.

Considering the time for selection, accuracy, and efficiency of evaluation functions, the recommendable upper boundary for the frequency range is between 0.15 and 0.3. This value is obviously larger than the expected value 0.075 in Figure 4. It is partly because most of our patterns had fewer squares than Buro’s (1998).
10.5
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0

Figure 12. Efficiency and accuracy of evaluation functions (55 and 60 discs).

8. Concluding Remarks

In this paper, we described a method of constructing accurate evaluation functions by using only the specifications of a target game and a set of training positions, which is crucial in constructing a general game player. Experiments on Othello revealed that a combination of pattern generation using logic and a lightweight pattern selection could efficiently search for and identify useful patterns. The method actually constructed accurate evaluation functions. The accuracy was by far superior to the evaluation functions generated by existing general methods, and was comparable (although slightly worse) to that of Buro’s (2002) which is part of a specialized Othello program.

Our intended future work aims at demonstrating the generality of the approach proposed here on other games, such as Shogi, where patterns with variable shapes are needed, and also at improving the efficiency of the generated evaluation functions in order to investigate total game-playing performance. The development of selection criteria taking efficiency into account seems promising, though investigations into their impact on accuracy would be required. It would also be challenging to develop a general method that introduces game-specific optimizations, including the use of patterns in fixed shapes, through an analysis of domain theory.

Acknowledgments

The authors would like to express their appreciation to the anonymous referees, whose comments led to significant improvements in this paper.
Automated Identification of Patterns in Evaluation Functions

References


AN EVALUATION FUNCTION FOR THE GAME OF AMAZONS

J. Lieberum
Mathematisches Institut, Univ. Basel, Rheinsprung 21, CH-4051 Basel
lieberum@math.unibas.ch, http://www.math.unibas.ch/~lieberum/

Abstract Amazons is a fascinating game that shares properties of chess and Go. We have written a computer program that plays Amazons. This paper reveals the secret of this program: its evaluation function. We describe it by explicit formulas, mention the ideas and goals behind these formulas, and discuss possible refinements. By analysing a tournament game of AMAZONG against the former computer world champion 8QP we illustrate how the new features of our evaluation function can lead to victory.

Keywords: Amazons, evaluation function, AMAZONG

1. Introduction

Amazons is a many-faceted game. The game set typically used to play Amazons is a draughts board of size $10 \times 10$, four white and four black chess Queens (called amazons), and a supply of Go pieces of one colour (called arrows). The starting position and a first move by White are shown in Figure 1. Each move consists of two steps: (1) the player chooses an amazon of the own colour and moves it like a chess Queen diagonally, vertically, or horizontally; the length of the move is up to the player as long as no obstacle (another amazon or an arrow) blocks the way; (2) this amazon has to throw an arrow. Arrows also move like chess Queens. They stay at their destination square for the rest of the game and are represented by black squares (in this paper) (see Figure 1 right). The players move alternately until one player can no longer move. This happens after at most 92 moves. The player who made the last move wins the game. White’s advantage of making the first move can be compensated by allowing Black to pass $n$ times (e.g., $n = 4$).

We first heard about amazons at a workshop on combinatorial game theory at MSRI in July 2000. We were fascinated by the deepness and subtlety of ‘simple’ positions in Amazons that have been analysed by Berlekamp (2000), Snatzke (1996, 2002), Müller and Tegos (2002). Inspired by discus-
Figure 1. One good first move out of 2176 possible ones.

sessions with Müller about his computer program ARROW and our own experiences of playing Amazons we started to write the computer program AMAZONG. After two years of successive improvements, AMAZONG has won the Amazons tournament at the seventh Computer Olympiad in Maastricht in July 2002. The reader is invited to play against the java applet AMAZONG at http://www.math.unibas.ch/~lieberum/amazong/amazong.html.

The general design of our program with a special focus on selective search has been presented in talks at the Universities of Jena and Edmonton (Lieberum, 2002). This paper complements these talks and concentrates on AMAZONG’s evaluation function that causes its characteristic style of play, clearly distinguishes it from other programs, and is probably its main strength.

2. The Different Phases of an Amazons Game

AMAZONG distinguishes three phases of the game: (1) the opening at the beginning of the game, (2) the filling phase at the end of the game, and (3) the main game that consists of everything else.

The opening in Amazons is the greatest challenge for computer programs. The reason is fourfold: (1) the absence of opening theory, (2) a branching factor of more than 1000, (3) many situations with more than 20 reasonable moves, and (4) the need for calculating deep variations. At this moment human play is still superior to computers in the opening. At the Computer Olympiad in Maastricht in 2002 AMAZONG made a random choice of the first move out of three possibilities and then started to play according to the results of a selective 5- or 6-ply search. Meanwhile, the opening book has grown to a machine generated database containing more than 30,000 moves following ideas of Lincke (2001). However, the benefit of opening books is limited in Amazons because of the enormous complexity of the game.

The filling phase consists of those positions where each empty square on the board can be reached by at most one player by some sequence of moves.
In most games this happens after approximately 50 moves. The filling phase includes positions with completely decomposed boards, meaning that amazons of different colours are separated by arrows. Then the outcome of the game can be determined by counting the number of moves left to each player. Although this problem is NP-hard (Buro, 2001), it is not difficult to play correctly in most positions that show up in real games on a board of size $10 \times 10$. Typically, the players stop to play and agree on the outcome of the game when the filling phase starts.

Two examples of positions from the filling phase are illustrated in Figure 2. In the position on the left side of Figure 2 it seems that White has access to two empty squares, but he has to cut off one of the empty squares with his next move. Therefore this shape is called a defective territory. The position on the right side of Figure 2 is called zugzwang because it seems that Black has access to three empty squares, but if Black has to move before White does, then Black can only use two of the three empty squares. AMAZONG already tries to evaluate defective territory and many Zugzwang situations correctly before the filling phase starts. However, these parts of AMAZONG’s evaluation function are still far from being perfect. They will not be discussed here.

![Figure 2. Defective territory and zugzwang.](image)

Since in most Amazons games the opening book covers only the first few moves, one has to deal with many different situations in the main game until the filling phase begins and the outcome of the game becomes clear. One possible parameter which could help to choose an appropriate strategy in each situation is the number of moves played so far. AMAZONG uses a different parameter to choose its strategy. This will be discussed in the next section.

### 3. Territorial and Positional Evaluation

The goal of Amazons is to have access to more empty squares in the filling phase than the other player. When player $j$ ($j \in \{1, 2\}$, player 1 is White) has exclusive access to a region of $n$ squares, we count these squares as $n$ secure points of territory of player $j$. When both players can reach a square by some sequence of moves, it is more complicated to predict which player will eventually shoot at that square. For this purpose AMAZONG uses heuristics based on the following ways to measure distances on an Amazons board.
Define the distance \( d_1(a, b) \) of two squares \( a \) and \( b \) as the minimal number of chess Queen moves needed to go from \( a \) to \( b \). When there is no path, let \( d_1(a, b) = \infty \). Similarly, define the distance \( d_2(a, b) \) as the minimal number of chess King moves needed to go from \( a \) to \( b \). Obviously, we have \( d_1(a, b) \leq d_2(a, b) \). The distances of player \( j \) from square \( a \) are then given by

\[
D_1^j(a) = \min \{ d_i(a, b) \mid \text{the square } b \text{ is occupied by an Amazon of player } j \}.
\]

Figure 3 (left) is an example of \( D_1^j(a) \). The upper left corners of empty squares contain the values \( D_1^j(a) \) and the lower right corners contain the values \( D_2^j(a) \). Figure 3 (right) is an example of \( D_2^j(a) \).

\[\text{Figure 3. The minimal distances } D_1^j(a).\]

All Amazons programs seem to use \( D_1^j \) in one or another way (Hashimoto et al., 2001). The idea behind the definition of \( D_1^j \) is that \( D_1^1(a) < D_2^2(a) \) indicates that player 1 has better access to the square \( a \) than player 2. One heuristic for estimating the territory of player 1 is to assume that player 1 will eventually shoot to all squares \( a \) with \( D_1^1(a) < D_2^2(a) \). This heuristic works very well shortly before and in the filling phase. A problem of \( D_1^j \) at the beginning of the game is that a single amazon of player \( j \) in the centre can cause low values of \( D_1^j \) on the whole board, but player \( j \) cannot move the amazon into all directions at once. Here \( D_2^j \) comes in. One advantage of \( D_2^j \) is its locality: often amazons have to fulfill a certain task at their position like guarding the territory in their neighbourhood. Then a large value of \( D_2^j(a) \) indicates that player \( j \) cannot move towards the square \( a \) without causing a positional damage, despite a possibly low value of \( D_1^j(a) \). Another advantage of \( D_2^j \) over \( D_1^j \) is that it behaves more stable when the other player moves and shoots, especially when there are just
a few arrows. This makes $D^j_2$ useful for long-term estimates and will stabilise the evaluation function in the beginning of the game.

We use $D^j_1$ to assign local evaluations between $-1$ and $1$ to each empty square. Positive values indicate an advantage of player 1. Then we sum these numbers over all empty squares in order to transform the local evaluations into global ones. One possible formula for global evaluations $t_1, t_2$ is given by

$$t_i = \sum_{\text{empty squares } a} \Delta(D^1_i(a), D^2_i(a)),$$

where

$$\Delta(n, m) = \begin{cases} 
0 & \text{if } n = m = \infty \\
\kappa & \text{if } n = m < \infty, \\
1 & \text{if } n < m, \\
-1 & \text{if } n > m,
\end{cases}$$

and $-1 < \kappa < 1$ is a constant with $(-1)^j\kappa \leq 0$ when player $j$ is to move. The value $|\kappa|$ estimates the advantage of moving first when the distances of both players to an accessible square agree. We have made good experiences with $|\kappa| \leq 1/5$, but some fine-tuning is necessary after each modification of the evaluation function. We optimise the choice of $\kappa$ frequently in order to obtain a low volatility of the evaluations during iterative deepening. This should help to avoid odd-even effects and supports aspiration search with narrow $\alpha-\beta$-windows. (Marsland, 1986).

A program that uses the territorial evaluation $t_1$ as its evaluation function plays already quite reasonably, especially shortly before the filling phase. In contrast to that the value $t_2$ is useful in the beginning of the game but becomes less significant as the game goes on. The evaluations $t_i$ share the drawback that they do not take into account that large values of $D^j_1(a) - D^j_1(a)$ are better for player 1 than small values. Therefore, other local evaluations than $\Delta$ seem to be important, too. The generic approach is to replace $\Delta(n, m)$ by some array of parameters and then to optimise these parameters. We have made good experiences with the choices

$$c_1 = 2 \sum_{\text{empty squares } a} 2^{-D^1_1(a)} - 2^{-D^2_1(a)},$$
$$c_2 = \sum_{\text{empty squares } a} \min(1, \max(-1, (D^2_2(a) - D^1_2(a))/6)).$$

Notice that in $c_1$ the local advantage $(D^j_1(a), D^j_2(a)) = (1, 2)$ is rewarded by 0.5 points for player 1, $(2, 3)$ by 0.25 points, $(1, 3)$ by 0.75 points, and squares $a$ with $(D^j_1(a), D^j_2(a)) = (n, n)$ contribute 0 points. Other tuples are of minor
practical importance for $c_1$. In contrast to $c_1$, $c_2$ depends only on $D_2^2(a) - D_2^1(a)$ and only large differences indicate a clear advantage of one player.

Now we have to combine the values $t_i$ and $c_i$ into one evaluation function. A weighted sum with static weights does not seem to be appropriate for this because the importance of the values $t_i$ and $c_i$ varies during the game. Therefore, we should first try to compute the expected number $W$ of moves needed until the filling phase starts. Instead of trying to estimate $W$ directly we simply define

$$ w = \sum_a 2^{-|D_1^1(a) - D_2^2(a)|}, $$

where we sum over all empty squares $a$ with $D_1^1(a) < \infty$ and $D_2^2(a) < \infty$. Obviously, we have $w = 0$ if and only if the position belongs to the filling phase and typically $w$ decreases with the number of moves played. Therefore, we expect that a good estimate of $W$ will be some function of $w$. For our purposes, $w$ is just as good as $W$. Now define an evaluation $t$ as

$$ t = f_1(w)t_1 + f_2(w)c_1 + f_3(w)c_2 + f_4(w)t_2, $$

where $(f_i)_i$ is a partition of 1 (meaning $0 \leq f_i(w)$ and $\sum_i f_i(w) = 1$). The exact form of the functions $f_i$ is a problem of parameter optimisation. Our choice of $f_1$ has been guided by the observation that $t_1$ becomes more and more important during the main game and gives very good estimates of the expected territory shortly before the filling phase. Hence, $f_1$ is monotonously decreasing and satisfies $f_1(0) = 1$. The counterpart of $t_1$ is $t_2$. It rewards balanced distributions of the own amazons on the board or helps to hinder the other player from reaching such a distribution. This is most important at the beginning of the game. The values $c_1$ and $c_2$ allow to detect finer properties of the position than $t_1$ and $t_2$ alone, because they depend on the quality of local advantages. They support good positional play in the opening and a smooth transition between the beginning and later phases of the game. This is most evident for $t_1$ and $c_1$: while at the end of the game only $t_1$ counts, $c_1$ rewards moves in earlier phases of the game that replace clear local disadvantages by small disadvantages and small advantages by clear advantages.

4. Mobility of Individual Amazons

AMAZONG is trying to enclose amazons of the other player inside of small regions at the beginning of the game. Compared to other computer programs, this is AMAZONG’s main strength. In this section we present a modification of the evaluation function $t$ (see Section 3) that is responsible for this behaviour.

Enclosing amazons typically does not cause an appropriate change of $t$ (and especially of $t_1$) in the beginning of the game. This can be explained as follows:
when a single amazon $A$ of player 1 is enclosed in some small region of $n$ points, then the Amazons board is divided into two parts: the inside and the outside of that region. Player 1 has exclusive access to the territory on the inside. This contributes $n$ points to $t$. On the outside, some active amazons of player 1 might overshadow the missing influence of $A$ in $D_1^1$. In addition, some amazons of player 2 that have helped to enclose $A$ might not be in optimal positions but often have a large potential to improve their positions. The problem that $A$ cannot reach the outside for the rest of the game is not reflected in the computation of $t$. The disadvantage of the enclosed amazon often starts to affect $t$ several moves later. Then it is too late. Therefore, a correction term $m$ is needed to take into account the mobility of individual amazons. Since active amazons can overshadow bad positions of passive amazons in the evaluation function $t$ it seems more important to punish passive and enclosed amazons than to support active amazons in this correction term. To compute $m$ quickly, consider first the number $N(a)$ of empty squares that can be reached from $a$ by a single move of a chess King. The numbers $N(a)$ can be updated incrementally during the search inside of functions $doMove$ and $undoMove$. For an amazon $A$ of player $j$ on the square $a$, let

$$\alpha_A = \sum_b 2^{-d_2(a,b)} N(b),$$

where we sum over all squares $b$ with $d_1(a, b) \leq 1$ and $D_{3-j}^1(b) < \infty$. When $\alpha_A = 0$ we say that the amazon $A$ is enclosed. Examples of the values $N(a)$, $\alpha_A$ and of enclosed amazons are shown in Figure 4. For example, for the white amazon $A$ in the upper left corner of the figure on the left, we compute $\alpha_A = 7 + 6 + 5 + 3 + 3 + (5 + 4 + 7 + 4)/2 + 5/4 = 35.25$. The two white amazons in the lower right corner in this figure are enclosed.

![Figure 4](image_url) Neighbours $N(a)$ of empty squares $a$ and the values $\alpha_A$. 
We have learned in discussions with experienced amazons players that at the beginning of a game on a board of size $10 \times 10$ enclosed amazons should be punished by a malus of at least 10 points. In general, we use $w$ from the last section to define

$$m = \sum_{\text{amazons } D \text{ of player 2}} f(w, \alpha_B) - \sum_{\text{amazons } A \text{ of player 1}} f(w, \alpha_A)$$

for a suitable function $f \geq 0$. The exact choice of $f$ is the hardest optimisation problem in our evaluation function $t + m$, so we restrict our description to the properties of $f$ that did not change during our experiments: $f$ satisfies $f(0, y) = 0$ and $\frac{\partial f}{\partial x}(x, y) \geq 0$ because the longer an amazon is enclosed before the filling phase starts the bigger is the disadvantage. Furthermore, $f$ satisfies $\frac{\partial f}{\partial y}(x, y) \leq 0$ because a low value of $\alpha_A$ corresponds to a passive position of the amazon $A$. The last dependence is not linear. We have made good experiences with functions $f$ that satisfy $2f(w, 5) < f(w, 0)$. This can be explained as follows: $\alpha_A \approx 5$ indicates that the amazon $A$ is almost enclosed. However, there is a big difference between an enclosed and an almost enclosed amazon. The other player possibly has to move an own amazon $B$ to an unfavourable square to prevent $A$ from escaping. The resulting change of $t$ then has to be compensated for by $m$. In addition, the task of guarding $A$ makes the amazon $B$ less mobile.

The big difference between enclosed and almost enclosed amazons can be seen on the right side of Figure 4: White can enclose the black amazon $B$ with $\alpha_B = 1$ in his next move, but then Black can reply by enclosing the white amazon, too. Similarly, the task of guarding the white amazon in the upper left corner puts the black amazon $B$ with $\alpha_B = 21$ in danger of getting enclosed.

5. Comparison between $t_1$ and $t + m$

In this section we compare our evaluation function $t + m$ with $t_1$ by using the game AMAZONG vs. 8QP played at the 7th Computer Olympiad in Maastricht. The position after 26 moves in this game is shown in Figure 3. AMAZONG won the game by 8 points, mainly due to the enclosed black amazon in the upper left corner. Figure 5 shows how $t_1$ and the different components of $t + m$ varied during the game.

In this diagram the values $t_i$ are computed using $|\kappa| = 0.1$. The lines corresponding to $t_1$, $t + m$ and $m$ are clearly visible in the diagram. On move 13 White enclosed the black amazon which causes the maximum of the dashed line corresponding to $m$. Notice that at this point the evaluation $t + m$ predicts the outcome of the game very well and differs from $t_1$ by more than 18 points. Then $t + m$ and $t_1$ become more and more related and finally coincide when the filling phase is reached.
As expected, the values $c_2$ and $t_2$ are more stable than $c_1$ and $t_1$. In addition, $c_2$ and $t_2$ are positive in almost all positions of the game. This indicates that the evaluation function of SQP does not consider King move distances. Therefore, SQP puts up no resistance against AMAZONG maximizing these components of $t + m$.

6. Refinements and Outlook

Consider positions with regions that are (almost) separated by arrows. How much is it worth when one player has a majority of amazons inside of such a region? Instead of looking for a general answer to this difficult question, we simply observe that the territorial evaluation $t$ has the tendency to underestimate the advantage of the majority. A possible correction term of $t$ could take into account the distances between each empty square and each amazon. However, the computations of these values would take almost four times longer than the computations of $D_1^j(a)$. Therefore, it seems more appropriate to compute only the numbers of amazons $A_{j
u}$ of player $j$ on squares $b_{\nu}$ that satisfy $d_i(a, b_{\nu}) = D_1^j(a)$. These numbers can be computed efficiently together with $D_1^j(a)$. They are useful as additional inputs of refined definitions of $c_i$ and $t_i$. In addition to these corrections, the disadvantage of having a majority of amazons in a small region early in the game should be reflected by $m$. This situation is not treated correctly by $m$ because when amazons of both players are inside of one region the involved amazons are not considered as being enclosed.

A second refinement concerns the distribution of amazons on the board. In the opening it is desirable (especially for Black) to reach a position with exactly one amazon in each corner of the board. The distances from such a distribution can be used to improve the evaluation function in the opening phase.
In some experiments, we weighted squares in the computations of $c_i$ and $t_i$. The weights depended on $w$ and the distance of the square from the centre of the board. It is difficult to assess the importance of this third refinement.

A fourth idea for improvements is to repeat the constructions of Section 3 for other distance functions such as $d_1 + d_2$ or $2d_1 + d_2$ (or estimates of these distances that can be computed more efficiently). One has to decide very carefully how many different distance functions one should use, because each additional distance function slows down evaluations considerably.

The biggest weakness of our evaluation function seems to be the underestimation of large territorial frameworks at the beginning of the game. Our hope (potential fifth refinement) is to incorporate ideas from Lorentz (2002) to overcome this weakness. It is difficult because in many situations one has to make a choice between two plans that are often incompatible: (1) chasing and enclosing amazons or (2) building large territorial zones. The decision which plan is the more promising one in an actual position is a challenge for the next generation of Amazons programs.

References


Abstract

Opponent-Model search is a game-tree search method that explicitly uses knowledge of the opponent. There is some risk involved in using Opponent-Model search. Both the prediction of the opponent’s moves and the estimation of the profitability of future positions should be of good quality and as such they should obey certain conditions. To investigate the role of prediction and estimation in actual computer game-playing, experiments with Opponent-Model search were performed in the game of Bao. After five evaluation functions had been generated using machine-learning techniques, a series of tournaments between these evaluation functions was executed. They showed that Opponent-Model search can be applied successfully, provided that the conditions are met.

Keywords: opponent models, search, evaluation functions, Bao

1. Introduction

This contribution investigates under what conditions the usual form of Opponent-Model search (OM search) can be made successful. To understand the matter we provide a condensed introduction to OM search in Section 2. In Section 3 we give a brief overview of the family of mancala games to which Bao belongs and we describe the Bao rules. In Section 4 we explain how we obtained five evaluation functions for Bao. Section 5 gives the tournament setup and in Section 6 we present and discuss the results. The contribution ends with conclusions in Section 7.

2. Opponent-Model Search

OM search (Carmel and Markovitch, 1993; Iida, Uiterwijk, and Van den Herik, 1993; Carmel and Markovitch, 1998; Donkers, Uiterwijk, and Van den Herik, 2003) is a game-tree search algorithm that uses a player’s hypothesized model of the opponent in order to exploit weak points in the opponent’s search
strategy. The original formulation of the OM-search algorithm is based on three strong assumptions concerning the opponent and the player:

1. The opponent (called MIN) uses minimax (or an equivalent algorithm) with an evaluation function \( \mathcal{V}_{op} \), a search depth and a move ordering that are known to the first player (called MAX);
2. MAX uses an evaluation function \( \mathcal{V}_0 \) that is better than MIN’s evaluation function;
3. MAX searches at least as deep as MIN.

This OM search procedure prescribes that MAX maximizes at max nodes, and selects at min nodes the moves that MAX believes MIN would select. Below we provide a short technical description of OM search, its notation, the relations between the nodes in the search tree, and some hints for an efficient implementation. For an extensive description of OM search we refer to Donkers et al. (2003).

OM search can be described by the following equations, in which \( \mathcal{V}_0(\cdot), \mathcal{V}_{op}(\cdot) \) are the evaluation functions, and \( v_0(\cdot), v_{op}(\cdot) \) are the node values. Subscript ‘0’ is used for MAX values, subscript ‘op’ is used for MIN values.

\[
\begin{align*}
v_0(P) &= \begin{cases} 
\max_j v_0(P_j) & \text{max nodes,} \\
v_0(P_j), & j = \min \arg_i v_{op}(P_i) & \text{min nodes,} \\
\mathcal{V}_0(P) & \text{leaf nodes.}
\end{cases} \\
v_{op}(P) &= \begin{cases} 
\max_j v_{op}(P_j) & \text{max nodes,} \\
\min_j v_{op}(P_j) & \text{min nodes,} \\
\mathcal{V}_{op}(P) & \text{leaf nodes.}
\end{cases}
\end{align*}
\]

If \( P \) is a min node at a depth larger than the search-tree depth of the opponent, then \( v_0(P) = \min_j v_0(P_j) \).

2.1 Implementation

For a search tree with branching factor \( w \) and even fixed depth \( d \), OM search needs exactly \( n = w^{d/2} \) evaluations of \( \mathcal{V}_0(\cdot) \) to determine the root value, since the search strategy is as follows: in each max node all \( w \) children are investigated and in each min node, only one child is investigated (see Donkers, Uiterwijk, and Van den Herik, 2001). Because the OM-search value is defined as the maximum over all these \( n \) values of \( v_0(\cdot) \), none of these values can be missed. This means that the efficiency of OM search depends on how efficient the values for \( v_{op}(\cdot) \) can be obtained.

A straightforward and efficient way to implement OM search is by applying \( \alpha-\beta \) probing: at a min node it starts performing \( \alpha-\beta \) search with the opponent’s evaluation function (the probe), and thereafter it performs OM search with the
move that $\alpha$-$\beta$ search has returned; at a max node, it maximizes over all child nodes. The probes can be efficiently implemented by using an enhanced form of $\alpha$-$\beta$ search. Because for every min node, a separate probe is performed, many nodes are visited during multiple probes. (For example, every min node $P_j$ on the principal variation of a node $P$ will be probed at least twice.) Therefore, the use of transposition tables leads to a major reduction of the search tree.

The $\alpha$-$\beta$ probes at a min node $P$ and at each grandchild (min nodes $P_{jk}$) are not independent since the $\alpha$-$\beta$ value of $P$, $v_{op}(P)$, is necessarily equal to or larger than all $\alpha$-$\beta$ values of $P_{jk}$. This means that $v_{op}(P)$ can be used to reduce the window of the probes at the grandchild nodes by setting the $\beta$ parameter of the probe at $P_{jk}$ to $v_{op}(P) + 1$.

OM search assumes that MAX speculates on all min nodes about the move that MIN is going to choose. In deeper parts of the search tree, the prediction of MIN’s move is based on shallower $\alpha$-$\beta$ probes than higher in the tree. It could therefore be justified to speculate only in the upper portion of the search tree.

### 2.2 Risk in Opponent-Model Search

Although using knowledge of the opponent during search seems obvious and OM search looks like a reasonable approach, there are three different types of risk involved. If these risks are not taken seriously, OM search is bound to fail.

First, OM search does not take into account any uncertainty about the opponent: the reasoning by the algorithm assumes perfect knowledge in the above sense. Since perfect knowledge of the opponent is hardly available in reality, this is a strong assumption. When the knowledge of the opponent is not perfect, the algorithm can still be used, but this will cause a certain amount of risk, depending on the quality of the knowledge. This first kind of risk has been described extensively in Iida, Handa, and Uiterwijk (1995). (In Donkers et al. (2001) an extension of OM search is described that does include uncertainty: Probabilistic Opponent-Model search.)

In Donkers et al. (2003), a second kind of risk in using OM search is investigated. It appears that even when MAX has perfect knowledge of MIN’s evaluation function, using OM search may be unwise: when MAX makes a large overestimation of the profitability of a certain position while MIN is judging it correctly, then MAX is possibly attracted to that position. A condition that should prevent this from happening is called admissibility of the pair of evaluation functions: MAX should not overestimate a position that MIN not also overestimates.

A third kind of risk in using OM search (introduced in this contribution) is as follows. Perfect knowledge of the opponent’s evaluation function is not equal to a perfect prediction of the opponent’s moves. This is caused by the difference (normally one ply) in search depth between a player’s prediction of
the opponent’s move and the actual search that the opponent uses at the next move.

For OM search to be successful, the effects of these risks should be alleviated. In a series of experiments on the game of Bao, we investigate what the influence is of a good prediction of the opponent’s moves and what the influence is of a better estimation of the own profitability of positions. Moreover, we study the effect of risk management. We assume perfect knowledge of the opponent’s evaluation function but admissibility is not guaranteed.

3. The Mancala Game Bao

In large parts of the world, board games of the mancala group are being played in completely different versions (cf. Murray, 1952; Russ, 2000). Whatever the case, most mancala games share the following five properties:

(1) the board has of a number of holes, usually ordered in rows;
(2) the game is played with indistinguishable counters (also called pebbles, seeds, shells);
(3) players own a fixed set of holes on the board;
(4) a move consists of taking counters from one hole and putting them one-by-one in subsequent holes (sowing), possibly followed by some form of capture;
(5) the goal is to capture the most counters (for Bao it is to immobilize the opponent).

Mancala games differ in the number of players (1, 2 or more), the size and form of the board, the starting configuration of the counters, the rules for sowing and capturing, and in the way the game ends. The games of the mancala group are known by many names (for instance Wari, Awele, Bao, Dakon, and Pallankuli). For an overview of different versions and the rules of many mancala games, we refer to Russ (2000).

Among the mancala games, (Zanzibar) Bao is regarded as the most complex one (De Voogt, 1995). This is mainly due to the amount of rules and to the complexity of the rules. Bao is played in Tanzania and on Zanzibar in an organized way. There exist Bao clubs that own the expensive boards and that organize official tournaments.

The exact rules of the game are given in, for example, De Voogt (1995). Below, we summarize the properties that discriminate the game from the more widely known games Kalah and Awari.

Bao is played on a board with 4 rows of 8 holes by two players, called South and North, see Figure 1. Two square holes are called houses and play a special part in the game. There are 64 stones involved. At the start of the game each player has 10 stones on the board and 22 stones in store. Sowing only takes place on the own two rows of holes. The direction of sowing is not fixed. At
the start of a move, a player can select a direction for the sowing (clockwise or anti-clockwise). During sowing or at a capture, the direction can turn at some point. This is dictated by deterministic rules.

If a capture is possible then it is obliged in Bao. This means that a position is either a capture position or a non-capture position. Captured counters do not leave the board but re-enter the game. Counters are captured from the opponent’s front row. These counters are immediately sown in the own front row. This implies that the game does not converge like Kalah and Awari.

Moves are composite. If at the end of a sowing, capture is possible, the captured counters are sowed immediately at the own side of the board. This second sowing can again lead to a new capture followed by a new sowing. If a capture is not possible, and the hole reached was non-empty, all counters are taken out of that hole and sowing continues. This procedure goes on until an empty hole is reached, which ends the move.

Moves can be endless because in a non-capture move, sowing can go on forever. The existence of endless moves can be proven theoretically (Donkers, Uiterwijk, and De Voogt, 2002). In real games, moves that take more than an hour of sowing also occasionally occur, but players usually make small mistakes during sowing or simply quit the game. So, real endless moves never lead to endless sowing.

Bao games consist of two stages: in the first stage, stones are entered one by one on the board at the start of every move. In the first stage, a game ends if the player to move has no counters left in the front row. As soon as all stones are entered, the second stage begins and a new set of rules applies. In the second stage, a game ends if the player has no more than one counter in any hole of both rows. A draw is not defined in Bao. Note that the goal of Bao is not to capture the most stones, but to immobilize the opponent.

**Figure 1.** Opening position of Bao.
In Donkers and Uiterwijk (2002), an analysis of the game properties of Bao is provided. The state-space complexity of Bao is approximated to be $1.0 \times 10^{25}$, which is much higher than those of Awari ($2.8 \times 10^{11}$) and Kalah ($1.3 \times 10^{13}$). The shortest game possible takes 5 ply, but most games take between 50 and 60 ply because they end (soon) after the start of the second stage. The maximum number of moves possible at any position is 32, but the average number of possible moves varies between 3 and 5, depending on the stage of the game. Forced moves occur quite often. The average game length ($d$) and branching factor ($w$) are normally used to estimate the size of a game tree that has to be traversed during search ($w^d$). For Bao the estimate is roughly $10^{34}$. This number together with the game-tree complexity ($10^{25}$) places Bao in the overview of game complexities above checkers and in the neighbourhood of Qubic (Van den Herik, Uiterwijk, and Van Rijswijck, 2002).

4. Generating Evaluation Functions for Bao

In order to conduct the OM-search experiments, we created 5 different evaluation functions. We describe them below. (For operational reasons (see Section 5) we would like to have them ordered in increasing quality with respect to the strength of the resulting players.)

The first two evaluation functions were created by hand. The first one, called MATERIAL, simply takes the difference in the number of stones on both sides of the board as the evaluation score. The second hand-made evaluation function is called DEFAULT. This function incorporates some rudimental strategic knowledge of Bao. For instance, it is good to have more stones in your back row since this increases the mobility in the second stage of the game. The function awards 3 points to stones in the front row, 5 points to stones in the back row, and 5 additional points to opponent stones that can be captured. If the own house is still active, 200 extra points are given. The total score of the position is the score for MAX minus the score for MIN. There is a small asymmetry in this function: if MAX can capture MIN’s house $100$ points are rewarded, but if MIN can capture MAX’s house, only $50$ points are subtracted. This asymmetry is intended to produce a more offensive playing style.

The third evaluation function was created by using a genetic algorithm (Holland, 1975). The evaluation function was represented by an integer-valued chromosome of 27 genes: one gene for the material balance, one gene per hole for the material in the own back and front row, one gene per hole in the front row for capturing, one gene for an active house, and another gene for capturing the opponent’s house. The total score of a position was the score for the player minus the score for the opponent. The fitness of a chromosome was measured by the number of games out of 100 that it won against a fixed opponent. In these matches, both players used $\alpha$-$\beta$ search with search depth 6. The genetic-
Opponent-Model Search in Bao: Conditions for a Successful Application

algorithm parameters were as follows: the population size was 100, only the 10 fittest chromosomes produced offspring (using a single-point crossover), the mutation rate is 5 per cent for large changes in a gene (i.e., generate a new random number for the gene) and 20 per cent for minor changes (i.e., altering the value of a gene slightly). The genetic algorithm was continued until no improvement occurs anymore. We conducted three runs: in the first run, the opponent was DEFAULT. In the second and third run, the opponent was the winner of the previous run. The name of the resulting evaluation function is GA3.

Thereafter we used another machine-learning technique to create the fourth evaluation function, namely TD-Leaf learning (Baxter, Trigdell, and Weaver, 1998). This is a temporal-difference method that is specialized for learning evaluation functions in games. The evaluation function trained was a linear real-valued function with the same parameters as the genes in the chromosomes above, except that there were separate parameters for the two sides of the board. Batch learning is applied with 25 games per batch. The reinforcement signal used to update the parameters was the number of games in the batch that the player wins against a fixed opponent, in this case GA3. The search depth used in the games was 10. The $\lambda$-factor and the annealing factor both were set to 0.99. This produced our fourth evaluation function, called TDL2B.

The last evaluation function was also produced by TD-Leaf learning, but this time we used a normalized Gaussian network (NGN) as evaluation function, similar to way in which Yoshioka, Ishii, and Ito (1999) trained an evaluation function for the Othello game. The NGN had 54 nuclei in a 54-dimensional space. Every dimension correlated with a parameter in the previous evaluation function. The reinforcement signal was the number of games out of 25 won against a fixed opponent, being TDL2B. The search depth used in the games was 6, because the computation of the output for an NGN is relatively slow. No batch learning was applied here. The $\lambda$-factor was set to 0.8 and the annealing factor was set to 0.993. This evaluation function is called NGND6.

5. Experimental Set-up

We conducted seven different tournaments between five players that each used one of the five evaluation functions. We denote the players by the name of their evaluation function. All tournaments followed a double round-robin system: every player was matched against every other player, one time playing South and one time playing North. Each match between two players consisted of 100 games; hence each tournament counted 2000 games. In the tables below the results are reported in a special way (see the caption of Table 1). One reason is that we can easy read off from the table any improvement by an evaluation function. The games began at the start positions given in the Appendix and
were played to the end. To prevent problems with infinite moves, any move that involves the sowing of more than 100 stones was considered infinite and illegal. A position at which a player could only perform one of these long moves was a loss for that player.

In the first two tournaments, both players used \( \alpha-\beta \) search. In the other five tournaments, South used OM search with perfect knowledge of the opponent's evaluation function. North always used \( \alpha-\beta \) search with search depth 6. The search depth of South differed per tournament. No time restrictions were given. We used an implementation of OM search with \( \alpha-\beta \) probes and allowed only one ply of speculation. Since in Bao draws are not possible, and since we aimed to compare the performance of the different search algorithms used by South, the score of a match was just the number of games out of 100 that was won by South.

At every position at which South was to move, we also detected the move(s) that \( \alpha-\beta \) search would select for South. In this way we were able to count the number of times that OM search differed from \( \alpha-\beta \).

In the implementation of the \( \alpha-\beta \) probes for OM search we took care of the fact that (some of) the evaluation functions are asymmetric. The asymmetry implies that evaluating a position when South is MAX, is not the same as evaluating the same position when North is MAX and taking the negative of the value. Furthermore, we dealt with multiple equipotent moves for MIN: if MIN has multiple equal choices, MAX will select the move with the lowest value for \( v_0 \).

The exact set-up of each of the seven tournaments will be explained along with the results in the next section.

6. Results and Discussion

First tournament: \( \alpha-\beta \) plain — Table 1 gives the outcome of the first tournament. Both South and North used \( \alpha-\beta \) search with search depth 6. The table clearly shows that the evaluation functions differ in quality and that every following evaluation function is operationally better than any of the previous ones. (Since the size of each match is 100, the 95% confidence intervals are approximately plus/minus 10 per match and plus/minus 20 for the total scores.)

Second tournament: \( \alpha-\beta \) extended — The second tournament was a checking tournament. South was allowed to search two extra plies (8 instead of 6). The results are presented in Table 2. The table shows that all players profited from the increased search depth. Only the match of DEFAULT against GA3 was less fortunate for DEFAULT. This illustrates the poor quality of this evaluator.

Third tournament: OM plain — In the third tournament, South used OM search with one ply of speculation and with search depth 6. The results in Table 3 show that three players, MATERIAL, GA3, and TDL2B, profited from using
OM search, but that the two other players, DEFAULT and NGND6, did not profit and played worse than in the first tournament.

Fourth tournament: OM extended — South was using OM search with one ply of speculation as in the third tournament, but in this tournament it was allowed to search two ply deeper. The $\alpha$-$\beta$ probes were still restricted to depth 6. This means that South had better knowledge over the game than North, a situation that is comparable to the second tournament. Table 4 shows that South was not able to profit fully from the extra search depth. Although all players performed better than in the third tournament where they were given a depth of 6 ply, only MATERIAL played better than in the second tournament. This indicates that searching deeper for yourself in OM search is not sufficient for success.

Fifth tournament: OM with perfect opponent prediction — The fifth tournament gave South a different advantage: it was allowed to extend the $\alpha$-$\beta$ probes to depth 7. The search depth (for the own evaluation) was 6. In this way, South not only had perfect knowledge of the opponent’s evaluation function, but South could also predict almost perfectly what North would be doing in the next move. The search depth of the $\alpha$-$\beta$ probes (which was 6, because the probes started at depth 1) was namely exactly the same as the search depth of North. In the case of equal evaluated moves, South selected the move with the lowest own evaluation. This was not necessarily the move that North would play. Table 5 gives the results of this tournament. All players, except DEFAULT profited from this advantage, and played better than in tournament 1, albeit less good than in the second tournament in which they just searched deeper. The advantage also gave less good results than the advantage in tournament 4, except for player NGND6. From these results we can infer that knowing exactly the moves of the opponent does not help if the own judgement is too weak.

Sixth tournament: OM perfect — The sixth tournament combined the advantages of the fourth and fifth tournament for South. The search depth for the own

<table>
<thead>
<tr>
<th>S \ N</th>
<th>MATERIAL</th>
<th>DEFAULT</th>
<th>GA3</th>
<th>TDL2B</th>
<th>NGND6</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATERIAL</td>
<td>-</td>
<td>55</td>
<td>35</td>
<td>19</td>
<td>18</td>
<td>127</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>48</td>
<td>-</td>
<td>54</td>
<td>30</td>
<td>28</td>
<td>160</td>
</tr>
<tr>
<td>GA3</td>
<td>55</td>
<td>61</td>
<td>-</td>
<td>36</td>
<td>30</td>
<td>182</td>
</tr>
<tr>
<td>TDL2B</td>
<td>69</td>
<td>65</td>
<td>57</td>
<td>-</td>
<td>39</td>
<td>230</td>
</tr>
<tr>
<td>NGND6</td>
<td>79</td>
<td>73</td>
<td>75</td>
<td>60</td>
<td>-</td>
<td>287</td>
</tr>
</tbody>
</table>

*Table 1.* Results of the first tournament between 5 evaluation functions for Bao. Each cell shows the number of games won (out of 100) by South (the row) against North (the column). The column on the right shows the number of games won (out of 400) by each evaluation function when playing South.
Table 2. Results of the second tournament between 5 evaluation functions for Bao. Both sides use $\alpha$-$\beta$, but South searches 2 ply deeper (8) than North (6).

<table>
<thead>
<tr>
<th>S \ N</th>
<th>MATERIAL</th>
<th>DEFAULT</th>
<th>GA3</th>
<th>TDL2B</th>
<th>NGND6</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATERIAL</td>
<td>-</td>
<td>57</td>
<td>62</td>
<td>40</td>
<td>32</td>
<td>191</td>
</tr>
<tr>
<td>Default</td>
<td>71</td>
<td>-</td>
<td>52</td>
<td>49</td>
<td>34</td>
<td>206</td>
</tr>
<tr>
<td>GA3</td>
<td>80</td>
<td>75</td>
<td>-</td>
<td>62</td>
<td>49</td>
<td>266</td>
</tr>
<tr>
<td>TDL2B</td>
<td>86</td>
<td>76</td>
<td>69</td>
<td>-</td>
<td>57</td>
<td>288</td>
</tr>
<tr>
<td>NGND6</td>
<td>88</td>
<td>76</td>
<td>80</td>
<td>70</td>
<td>-</td>
<td>314</td>
</tr>
</tbody>
</table>

Table 3. Results of the third tournament between 5 evaluation functions for Bao. South uses OM search with perfect knowledge of the opponent’s evaluation function. The search depth is 6 for both sides.

<table>
<thead>
<tr>
<th>S \ N</th>
<th>MATERIAL</th>
<th>DEFAULT</th>
<th>GA3</th>
<th>TDL2B</th>
<th>NGND6</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATERIAL</td>
<td>-</td>
<td>57</td>
<td>50</td>
<td>30</td>
<td>24</td>
<td>161</td>
</tr>
<tr>
<td>Default</td>
<td>46</td>
<td>-</td>
<td>46</td>
<td>26</td>
<td>25</td>
<td>143</td>
</tr>
<tr>
<td>GA3</td>
<td>59</td>
<td>57</td>
<td>-</td>
<td>40</td>
<td>35</td>
<td>191</td>
</tr>
<tr>
<td>TDL2B</td>
<td>78</td>
<td>64</td>
<td>60</td>
<td>-</td>
<td>46</td>
<td>248</td>
</tr>
<tr>
<td>NGND6</td>
<td>71</td>
<td>58</td>
<td>66</td>
<td>61</td>
<td>-</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 4. Results of the fourth tournament between 5 evaluation functions for Bao. South uses OM search with perfect knowledge of the opponent’s evaluation function. The search depth is 8 for South, with $\alpha$-$\beta$-probes to depth 6, and the search depth is 6 for North.

<table>
<thead>
<tr>
<th>S \ N</th>
<th>MATERIAL</th>
<th>DEFAULT</th>
<th>GA3</th>
<th>TDL2B</th>
<th>NGND6</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATERIAL</td>
<td>-</td>
<td>60</td>
<td>64</td>
<td>49</td>
<td>39</td>
<td>212</td>
</tr>
<tr>
<td>Default</td>
<td>63</td>
<td>-</td>
<td>47</td>
<td>44</td>
<td>41</td>
<td>195</td>
</tr>
<tr>
<td>GA3</td>
<td>70</td>
<td>66</td>
<td>-</td>
<td>57</td>
<td>40</td>
<td>233</td>
</tr>
<tr>
<td>TDL2B</td>
<td>80</td>
<td>69</td>
<td>70</td>
<td>-</td>
<td>56</td>
<td>275</td>
</tr>
<tr>
<td>NGND6</td>
<td>84</td>
<td>68</td>
<td>71</td>
<td>59</td>
<td>-</td>
<td>282</td>
</tr>
</tbody>
</table>

evaluation was 8 for South and the $\alpha$-$\beta$ probes for the opponent extended to depth 7. The results in Table 6 show that the power of South was significantly increased. All players performed better than in tournament 1, and all players, except GA3 also played better than in tournament 2. The results of GA3 were only slightly less than in tournament 2.

Seventh tournament: OM with strict risk management — In the seventh and last tournament, South applied OM search with strict risk management. South only
deviated from the strategy that $\alpha-\beta$ search imposed if the move that OM search advised had the same Minimax value. The search depth was equal to the third tournament. Since it occurred relatively often in Bao that multiple moves at the same position had the same Minimax value, South did have some room to speculate. The results in Table 7 show that this approach was successful too.
All players performed better than in the first tournament and also better (or equally poor in case of TDL2B) than in the third tournament.

A summary — Table 8 summarizes the results of the seven tournaments. Each cell contains the final score of a tournament (400 games), playing South. On all rows the scores are increasing from left to right (except for the first two columns). This means that the order of quality for the evaluation functions indeed is as follows: (MATERIAL, DEFAULT) < GA3 < TDL2B < NGND5. The ordering of MATERIAL and DEFAULT is unclear, but both evaluation functions are poor. The table shows that if only the search depth is increased (4: OM extended) or only the prediction of the opponent is improved (5: OM perf. opp.), the results are not as good as just using $\alpha-$ with two additional ply of search. When both methods were combined, (6: OM perfect), the results were better. Furthermore, the table shows that using OM search with strict risk management (7: OM no risk) led to better results than using plain OM search and plain $\alpha-$.

Deviations — The last overview, in Table 9, provides insight into the number of times that OM search deviated from the $\alpha-$-search strategy. The table shows that searching more deeply for the own evaluation had a larger effect than searching more deeply for the prediction of the opponent. The table also shows that the number of deviations was larger in tournament 4 than in tournament 6. Since the results of tournament 4 were less good than the results of tournament 6, it seems that an incorrect prediction of the opponent leads to extra deviations that did not contribute to a positive outcome.

7. Conclusion

The experiments described in this paper are a follow-up to earlier experiments with OM search in other game domains. In Donkers et al. (2003) we described experiments in Lines of Action and in the chess endgame KQKR. The experiments in Lines of Action showed that OM search with evaluation functions of poor quality led to bad results. The experiments in the chess endgame KQKR
showed that OM search with a perfect evaluation function (i.e., an endgame database) for MAX can be useful, but the results are not conclusive.

The Bao experiments in this contribution were designed to identify those factors that influence the success or failure of OM search. Although the experiments were not encyclopedic and therefore did not produce firm qualifications of these factors, many effects are statistically significant. In all, the Bao experiments provide a good insight into the working of OM search. For instance, it appears that a combination of adequate opponent prediction and extended search depth is needed for good results. Of these two factors, the extended search depth seems to be more important than the good prediction. Moreover, the quality of the evaluation functions appears to be important for the effect of OM search. For plain OM search the results were not good for most of the players because the evaluation functions do not obey the admissibility requirement.

A generalisation of the results to other games leads to the statement that the search method can only be applied successfully when additional resources (e.g., search time) are available. The additional search time (in comparison with the opponent) must either be spent for the prediction of the opponent’s move, or for the risk management. If these additional resources are not available, OM search cannot with certainty be applied successfully.

In order to measure the effects of opponent prediction and extended search more precisely, the sample size should be increased and more game details should be analysed, such as the number of times that the predicted move differs from the move played by the opponent. Furthermore, a deeper study of the properties of the trained evaluation functions and the matches between players themselves might provide more background information. A final suggestion for future research is to investigate the possibilities for risk management more deeply since this seems a promising approach.
References


Appendix

The following table gives the 100 start positions used in the Bao experiments. The positions are generated by playing 10 random legal moves for every player from the official Bao opening position. Each row gives the contents of the holes of one position. The numbering of the holes is according to De Voogt (1995). The last two columns indicate whether South and North have an active house.
Kriegspiel is a chess variant invented to make chess more similar to real warfare. In a Kriegspiel game the players have to deal with incomplete information because they are not informed of their opponent’s moves. Each player tries to guess the position of the opponent’s pieces as the game progresses by trying moves that can be either legal or illegal with respect to the real situation: a referee accepts the legal moves and rejects the illegal ones. However the latter are most useful to gain insight into the opponent’s position. While in the past this game has been popular in research centres such as the RAND Institute, currently it is played mostly over the Internet Chess Club.

The paper describes the rationale and design of a Kriegspiel program to play the ending for King and Rook versus King. Such a kind of ending has been theoretically shown to be won for White, however no programs exist that play the related positions perfectly. We introduce an evaluation function to play these simple Kriegspiel positions, and evaluate it.

Keywords: Kriegspiel, Eastern rules, Western rules, metaposition

1. Introduction

The game of chess has been widely studied because it is a microcosm that mirrors decision making in real-world situations. However, a basic limit of chess as a field for studying decision making is that decisions by players have nothing to do with uncertainty in the sense in which the term is used in game theory, since the goal and the best strategy for each player can be computed easily and completely.

The game of Kriegspiel is a chess variant invented around 1896 to make chess more similar to real warfare. It involves incomplete information: both the premises and the consequences of a decision are partially unknown, thus it is considered a complex game because of the asymmetry in the knowledge available to the players as the game progresses. In fact, when a player makes an illegal move, from his failure he can infer data that cannot be inferred by
his opponent as well. Thus, in general, during a Kriegspiel game each player knows what he knows, but he does not know what his opponent knows.

Kriegspiel is a game interesting in several ways. First, it is based on the same rules as chess, but is has a completely different (and not well studied) theory.

It is a game of imperfect information, such as Poker. However Kriegspiel has no stochastic element, which makes it different from Poker. To play Kriegspiel well we have to use logic and the mathematics of probability.

At the moment there are no programs that play a reasonable Kriegspiel game. On the Internet Chess Club (ICC) a couple of programs are available, which are able to play Kriegspiel, however none of these programs is among the best players (on ICC there are several hundreds of Kriegspiel players, and every day they play hundreds of games).

We recall that a number of papers have studied some aspects of Kriegspiel or Kriegspiel-like games. Below we provide some instances of related work. Ferguson (1992, 1995) analyses the endings KBNK and KBBK, respectively. Ciancarini, DallaLibera, and Maran (1997) describe a rule-based program to play the KPK ending according to some principles of game theory. Sakuta and Iida (2000) describe a program to solve Kriegspiel-like problems in Shogi (Japanese Chess). Bud et al. (2001) describe an approach to the design of a computer player for a subgame of Kriegspiel, called Invisible Chess.

In this paper we explore some issues of the ending KR vs K in Kriegspiel. We aim to design a program that will be a prototype component of a multi-agent system able to play Kriegspiel. We describe how we have built such a component, and how we evaluate its behaviour, with the purpose to improve its playing ability.

This paper has the following structure. In Section 2 we describe the basic rules of Kriegspiel, including a study of its main variants. In Section 3 we introduce the theory of the KRvsK ending in Kriegspiel. In Section 4 we describe our search algorithm. In Section 5 we describe our evaluation function: it is specific for this ending, but in our knowledge it is the first time an evaluation function for playing Kriegspiel has been defined. In Section 6 we describe how we use a transposition table to support the search across a tree of metapositions. Finally, in Section 7 we evaluate our approach.

2. The Rules

Perhaps the lack of standard rules has been an obstacle to the diffusion of Kriegspiel as a research subject. In fact, there are several different sets of rules, basically classified into two families as Eastern rules (widespread in UK and Eastern US) and Western rules (widespread in Western US) (Pritschard, 1994; Li, 1994). The rules given by J.D. Wilkins in Williams (1950) have been used for years in the RAND Institute. The ICC rules are derived from the RAND...
rules. However, ICC managers introduced some variants which make the play over the Internet slightly more difficult.

A Kriegspiel player tries a move selected among the set of his pseudo-legal moves, including possible pawn captures. For instance, in the Diagram 1: possible tries for White are $\text{h2, a3, d2, e3, f4, g5, h6, d1, d2, e2, f2, f1, d5, dc5, de5.}$

The referee, who knows the list of legal moves for both sides, answers all tries with one of the following six messages.

**End of game** If the list of legal moves is empty the position is checkmate or stalemate, and the referee announces the corresponding finish.

**Move accepted** If the try is legal, the referee says “White moved” (or “Yes”) and gives no further information. We denote this situation also as “Silent referee”, because he gives no useful information.

**Illegal move** The try selected by White might be illegal on the referee’s board. For instance, in the position of Diagram 2 (as it is on the referee’s board) White could try $\text{h6.}$

The referee says “Illegal move” (or “No”) and White infers that either the diagonal to $\text{h6}$ is obstructed by an enemy piece, or the Bishop is pinned by a black major piece in $\text{a1}$ or $\text{b1}.$

**Impossible move** The message “Impossible move” is given when a player tries a move outside his set of pseudo-legal moves. In Diagram 2 an impossible try could be $\text{e3.}$

**Check** If a move is accepted and gives check, the referee announces the check and its direction (row, column, major diagonal, minor diagonal, Knight). In the example, the move $\text{d2}$ gets the answer “Check on major diagonal”.

**Capture** The referee announces all captures, but he says only on which square the capture takes place, and says nothing about the capturing or captured piece. In the example, the move $\text{f4}$ gets the answer “Capture on f4”.

![Diagram 1. Possible tries.](image)

![Diagram 2. The referee's board.](image)
This list describes the basic messages from the referee. However, in all Kriegspiel versions there is a special treatment of positions where captures by Pawns are possible (We continue numbering of messages).

**Are there any?** In the original set of rules (Eastern rules) a player could ask before each move “Are there any?”, intending “Are there any captures by my Pawns?”. The referee answers “No” if no capture is possible, or “Try!” if one or more captures are available. With RAND rules the referee announces before each move all possible pawn captures, naming the squares where they can take place. In the set of rules which is used on the Internet Chess Club (Western rules) the referee announces *before each move* how many pawn captures are available.

Diagram 3 shows the differences among the different set of rules.

**Eastern rules**: The referee says: “White to move”. White can choose to ask “Are there any?”; if in the above position White asks the question, the referee says “Try!”; White then has to try at least one capture out of three, namely ab4, cb3, or cd3.

**RAND rules**: *before* White moves the referee says to both players “*possible pawn capture on b4*”. White is not obliged to capture.

**Western rules, ICC**: *before* White moves the referee says to both players “*possible one pawn capture*”. White is not obliged to capture.

If now White moves his Pawn to c4,

**Eastern rules**: the referee announces: “*Black moves*”;

**RAND rules**: the referee announces: “*possible pawn captures on a3 and c3*”;

**Western rules, ICC**: the referee announces: “*possible two pawn captures*”.

We report these differences for completeness, but we also note that they are not important for endings without Pawns. More important when dealing with endings is the fact in the original form of Kriegspiel no 50-move rule is included; instead on ICC the 50-move rule is enforced.

As a final remark, we note that there are several other forms of Kriegspiel-like games, like Dark Chess, Invisible Chess, Stealth Chess, and others. They are all based on some form of invisibility. We plan to report the features of this family of games in a future paper.
3. KR vs K in Kriegspiel

The ending KR vs K in chess is won in at most 16 moves starting from any position. This chess ending is quite easy to study by brute force, because excluding symmetric positions only 28,000 positions have to be evaluated, as shown in Clarke (1977).

The ending KR vs K in Kriegspiel is also won. However according to H.A. Adamson who published some analysis in the magazine Chess Amateur in 1923 and 1926, it can take even 40 moves to give checkmate to Black. More recently, this ending has been studied by Leoncini and Magari (1980) and Boyce (1981). The studies proved that this ending is algorithmically won, i.e., White can force mate against any defense, even the most clairvoyant; there are, instead, several endings (e.g., KP vs K or KBB vs K) which are only probabilistically won, that is Black has a chance to draw (or, equivalently, if the referee suggests Black the right move) (cf. Ferguson 1992, 1995; Ciancarini et al., 1997).

Below we start with developing an algorithm for KRK. Therefore we define the notion metaposition. A metaposition is a position describing a set of positions: this can be done graphically. In our case we have diagrams with several black Kings, meaning that its position is uncertain. Subsequently we can evaluate how many KRK metapositions we have to deal with. The number of metapositions for this ending can be approximated by fixing the position of white pieces and considering the number of the ways to choose \( n \) BK’s positions among the remaining positions. If we assume as a worst case for White: \( \text{al} \) and \( \text{b1} \), we have 52 possible positions which are not controlled by White. The possible metapositions are then

\[
\sum_{1 \leq n \leq 52} \binom{52}{n} = 2^{52} - 1
\]  

For these positions, the reflections of the BK position with respect to the diagonal al to h8, as described in Bain (1994), do not decrease the numerical complexity of the problem. So we are not able to study this ending completely by brute force.

Diagram 4 shows a typical ending. This diagram shows a metaposition: the double black King means that White is not sure whether the black King is on a8 or on b8. Alas, he has to find the best (most rapid) route to checkmate.
White tries 1.\textit{c7}: (1) if the referee says “No” then White tries 1.\textit{a6} or 1.\textit{c2} then mate;

(2) if the referee says “Yes” then 2.\textit{a1} #.

In Diagram 5, White tries 1.\textit{c7}:

(1) if the referee says “Yes” then the BK is on a8 and \ldots \textit{a7} 2.\textit{d6} \textit{a8} 3.\textit{a6}#.

(2) if the referee says “No” then White plays 1.\textit{c7} and:

(2a) with a silent referee White identifies the black King on a8 and the mate is very simple;

(2b) if the referee says “check” then 2.\textit{d7}:

(2b1) if “No” the BK is on d8 then 2.\textit{c1} \textit{e8} 3.\textit{f1} \textit{d8} 4.\textit{f8}#;

(2b2) if “Yes” Black played 1\ldots \textit{b8} then 2.\textit{d7} \textit{a8} 3.\textit{c6} \textit{b8} 4.\textit{b6} \textit{a8} 5.\textit{c8}#.

A general algorithm for any position, in which White knows nothing about the BK whereabouts, is given in Leoncini and Magari (1980). The procedure includes several phases.

In the first phase White has to configure his own pieces as in Diagram 6.

The second phase consists of looking for the BK by moves like \textit{d2}, \textit{e2}, \textit{d3}, \textit{e3},\ldots, \textit{d8}, \textit{e8}:

if the referee never says “check” then the BK is in the left-hand halfboard, otherwise when a check occurs the BK is in the right-hand halfboard, and White’s task will be easier to fulfil. We assume the first hypothesis in the metaposition shown in Diagram 7.

Interestingly, Kriegspiel metapositions have been compared by Magari to probability waves as in Quantum Physics. According to such a metaphor, the black King is not a body with a precise position, but a wave, or a set of possibilities. The white King has to destroy such a wave entering it and reducing the freedom of the black King.
In the final position of Diagram 8 White mates with $\text{a8#}$.

If at any time the referee says “Illegal move”, White will find the BK earlier, and will be able to use his Rook to restrict further the space available to Black.

### 3.1 Exploiting the Referee’s Answers

In any KRK ending, when White has to try a move, there are three possible situations.

1. The referee’s answer is ‘silent’. This allows us only to update our reference board cleaning the squares around the WK and along the WR row and column.

2. The WR can check the BK, in that case the player updates his reference board and assumes that the BK possible position is on the WR row or column.

3. A try may be illegal because the WK tries to go in a square which is under attack or because the WR is going across an occupied square.

Assume we are in the situation shown in the leftmost position in Diagram 9. If White moves $\text{e3}$ we distinguish two cases: (1) the referee’s answer is ‘silent’ (second position) or (2) the referee says a check has occurred (third position). If White moves $\text{d5}$ two cases can be outlined too, with a ‘silent’ answer or with an ‘illegal’ answer. In the rightmost position we show the result of obtaining the answer ‘illegal’, since the case in which we get a silent answer is similar to the Rook’s one.
4. The Search Algorithm

When drawing out the search algorithm we are first led to a problem caused by the fact that the move is described only by the referee’s answer. This implies that the evaluation of a move can be made only with respect to the referee’s answers, using some probabilistic reasoning.

Considering for example a situation where the WK is on c2, the white Rook is on f2, and Black’s positions traced in the white player’s reference board are on a1, a2, a3, or e3 with a likelihood of $\frac{1}{4}$ each. If the WK moves to b1 and receives an ‘Illegal’ answer that move will be a good move, decreasing the uncertainty (leftmost metaposition in Diagram 10), but if the move receives a ‘silent’ answer he will achieve a state of danger, where the WR risks to be captured (rightmost position in Diagram 10). So White should not play such a move.

Our solution consists of making a first evaluation during the generation of the pseudo-legal moves considering both cases, either ‘illegal’ or ‘check’ and ‘silent’ answers, and inserting in the possible moves vector the one with the lowest value. In other words the player assumes the worst case and makes available to the search algorithm only one answer per move. In this manner
the number of moves we are handling becomes more similar to that of classic chess.

In Kriegspiel the player is in the dark about his opponent’s position so a minmax-like search cannot be executed in this context, unless we find how to represent all possible BK moves. Simply adding a new layer to the algorithm and calculating for each White’s legal move all possible BK positions and for each of those positions repeat the procedure in a minmax way, has an exponential cost that forces us to choose some alternatives.

The way we have chosen to represent the invisible BK on White’s reference board is to define a metaposition which is a set of possible BK squares with the same likelihood. Also, we define an uncertainty index as the count of the possible positions of a metaposition, as in Sakuta and Iida (2000). In some sense White has to play against an unspecified number of black Kings, that can move simultaneously. It is quite simple to define a metamove as a move from one metaposition to another metaposition. Playing a metamove corresponds to playing all the moves for each black position of the metaposition. This trick allows us to use an algorithm like minmax or similar, where we use a metamove generator. We represent a metaposition as an array of possible positions.

One distinctive aspect to note is that we are changing the meaning of search depth. It now refers only to White’s branching factor, since the generation of a metaposition from another involves the introduction of a single edge. Diagram 11 describes the state reached from a reference board where the BK is assumed to be on g2 or g5 with a likelihood of $1/2$ each.

**Diagram 11.** Representing metapositions with likelihood.

Figure 1 shows the pseudo-code describing the search algorithm.

The algorithm generates all legal white moves and for each resulting position it evaluates both possible referee answers using an evaluation function we will discuss later. So, for each possible position, it is able to distinguish between ‘check’ or ‘illegal’ and ‘silent’ answers and it marks the move with the worst case according to the value returned by the evaluation function. If it has reached
Search Algorithm (int depth) {

    generate the white’s legal moves $\Gamma$;

    for each moves $j \in \Gamma$
    {
        if(rook plays the move $j$)
            $j$.value=Min(evaluate($j$,check),evaluate($j$,silent))
        if(king plays the move $j$)
            $j$.value=Min(evaluate($j$,illegal),evaluate($j$,silent))
    }

    for each moves $j \in \Gamma$
    {
        if (depth! = 1) {
            makemove($j$);
            generate the opponent’s metamove;
            if(!CheckHash(depth−1,&value))
                $j$.value += Search(depth-1);
            else
                $j$.value += value;
            unmakemove();
            if ($j$.value > max)
                max=$j$.value;
        }
    }

    RecordHash(depth,max);

    return max;
}

Figure 1. The search algorithm.

the desired search depth it simply returns the max move’s value, otherwise it
plays each move and in each metaposition obtained it makes the metamove,
then it decrements the depth of search and it recursively calls itself; after that, it
retracts the move played and adds to the move’s value the vote which is returned
by the recursive call. Finally, it updates the max on that particular search depth.

A move’s value is modified during the path that the algorithm is analysing. If
we did not make such updates, a move would obtain a good vote even crossing
bad states, where, as an example, we run the risk of losing the Rook. Figure 2
shows the search tree which describes a hypothetical visit. The first evaluation
is on the right of the node and the updated value of the move is on the left; the
bold type indicates the best move.
If we did not add the static evaluation value to the recursive value, at the first depth, moves would respectively obtain $-524, 313$ and $272$; so the second move (which has a bad static value) would be chosen by the search algorithm, while the third move (which has the greatest static value) would be discarded.

5. The Evaluation Function

We will implement the evaluation knowledge using a weighted linear function, as follows:

$$Evaluate(S) = c_1 f_1(S) + c_2 f_2(S) + ... + c_5 f_5(S)$$  \hspace{1cm} (2)

where $c_1, c_2, .., c_5$ are constants and $f_1(S), .., f_5(S)$ are functions which set up the heuristic evaluation.

The first aspect we want to make sure of is to avoid having a position where the WR risks to be captured. For this reason the first boolean function $f_1(S)$ evaluates the possibility that the Rook is under attack, in that case it returns FALSE.
Once we are certain that the Rook is safe, we try to bring the two Kings closer. That means to let the WK patrol the board. Thus the second function $f_2(S)$ estimates the distance between the WK and all the possible BK positions, by considering the furthest one. The way we calculate the distance is the sum of columns and rows between the WK and the furthest BK. In Diagram 12 we show an example where this distance is 10.

Let us assume that it is White’s turn to move, so the BK certainly is on one of those quadrilateral regions with which the white Rook divides the board. The aim of the white player is therefore to reduce all the regions’ areas that contain the black Kings. Again the uncertainty about BK’s real position is a problem. The third function $f_3(S)$ estimates which one of the four regions holds the BK and tries to reduce its area. We define it as

$$f_3(S) = \text{EvalArea}(S) = c \cdot (a_1 + a_2 + a_3 + a_4)$$  \hspace{1cm} (3)$$

where $c \in \{1, 2, 3, 4\}$ is the value which traces the number of quadrilaterals that possibly contain the opponent’s King, and $a_i (i = 1, \ldots, 4)$ represents the number BK’s possible positions in each quadrilateral. As shown in Diagram 13, in the worst case where uncertainty is maximal, the function’s result is 180.

The fourth function $f_4(S)$ is a boolean function which evaluates whether the WR is on the squares around the WK, in that case it increases by one the move’s value.

The fifth function $f_5(S)$ considers good moves those that push the BK toward the board’s corner. For each positions, where the BK might be, $f_5(S)$ adds to the move’s value the correspondent value from the matrix, shown in the Figure 3.

It is useful to note that $f_3(S)$ function calculates a positive value, but in order to evaluate the best move we have to minimize this value.

The same remark on the others functions leads us to the following evaluation function:

$$\text{Evaluate}(S) = -420 + 840 \cdot f_1(S) - f_2(S) - f_3(S) + f_4(S) + f_5(S)$$
Figure 3. The simple numerical matrix used by $f_5(S)$.

where $c_1 = 840$ is a weight that gives $f_1(S)$ top priority.

We finally add, after the search algorithm, a function, which catches checkmate cases and consequently avoids playing moves to stalemate states.

6. The Transposition Table

Since during the search algorithm we would cross states of the board previously analysed, it is interesting to avoid to analysing them a second time. As we have seen the number of metapositions is extremely large and it is impossible to maintain each of them in memory. A natural solution to the comparison between the states involves creation of a signature value, typically using Zobrist (1970) keys.

We define a three-dimensional vector indexed on \{KNIGHT, ROOK\}, \{WHITE, BLACK\}, and on the number of squares; then we fill each element with a random 64-bit number. To create a Zobrist key for a metaposition, we set it to zero, then for each piece on the board we add it into the key via the XOR operator. The pieces can be either the Kings or the Rook, and the black King may appear several times.

This technique has the advantage of creating good hash keys, that are not related to the metaposition being keyed. If a single piece is moved, we obtain a value that is completely different. So, these keys do not collide often. Another good peculiarity is that we can manage Zobrist keys incrementally, improving the artificial player’s performance, as described by Moreland (2002).

We use the Zobrist keys to implement a transposition table, which is a large hash table that allows us to trace metapositions that we have met during the search. It is impossible to create a big data structure that includes all the metapositions, but in the event of collisions, i.e., when two states are mapped on the same vector’s element, we use the Zobrist keys to identify the correct one.
```c
CheckHash (int depth, int *value) {
    hash_element *hashpt = &table[(WRB.key % MHE)+MHE];
    if(hashpt->key == WRB.key)
        if(hashpt->depth >= depth) {
            value=hashpt->value;
            return TRUE;
        }
    return FALSE;
}

RecordHash (int depth, int max) {
    hash_element *hashpt = &table[(WRB.key % MHE)+MHE];
    hashpt->key = WRB.key;
    if(hashpt==NULL) hashpt->value = value;
    else if(hashpt->value > value)
        hashpt->value = value;
    hashpt->depth = depth;
}
```

*Figure 4.* Updating the hash table. WRB means White’s Reference Board and MHE is the Max number of the Hash Elements into the table.

In the Figure 1 we used two functions whose pseudo-code is shown in Figure 4. These two functions are used to store the elements into the transposition table and to load them from it.

The CheckHash function does the load operation. If the element previously stored is the one we have to analyse and it has been examined with a depth grater or equal to the required depth, then the element is loaded from the table.

The RecordHash function does the store operation. It inserts the key and the search depth into the table. When it is not saving a new element, it inserts the value only if this value is smaller than the previous one. That means that the metapositions are randomly divided into clusters.

### 7. Tests

We have executed a first test on 26,536 initial positions, randomly selected from the 175,168 legal positions of KRK endgame. Each initial position has the maximum uncertainty on White’s reference board, meaning that the BK has the maximum freedom in terms of possible squares. Black’s strategy always consists in playing the move that allows him to go away from the edge of the board.
This test shows that 95.6% of the games are won by White, while 4.4% is lost. In particular 75.9% of this percentage refers to a game that has been stopped for a loop, 24% is draw, and 0.1% is a stalemate, as shown in Table 1. The average number of moves needed to give mate is 36, and the worst game played has been 117 moves long.

In order to have a comparison, we executed a second test on all initial positions using the referee’s point of view, namely we play this ending using ordinary chess rules and our Kriegspiel evaluation function. During a match, if the game either begins at or goes across some positions previously played, the referee stops it and considers it won or lost, depending on the result of previous games.

In this test 99.5% of the games is won, which is not bad but it shows that our evaluation function is not perfect for ordinary chess. We show the entire results in Table 2 and in Figure 6 we show the sets of won games and the number of moves needed during the second test.
Below we show how our program deals with the position of Diagram 5. If we assume that the BK is on a8, the program plays efficiently (in the scores, I means that the referee says illegal, C means check): 1. d6c7 a8b7{I} a8a7 2. d7d6 a7b6{I} a7b7{I} a7a6{I} a7a8 3. d6a6{C} 1-0 {White mates}

If we assume that the BK is on c8, the program actions are also very effective, as it achieves the checkmate in 2 moves: 1. d6c7{I} d7f7 c8d7{I} c8c7{I} c8b7{I} c8d8 2. f7f8{C} 1-0 {White mates}

Let us assume that the black King is on b8. After playing c7 and receiving an ‘illegal’ answer, the program plays less precisely f7, and then it takes 23 moves to mate: 1. d6c7{I} d7f7 b8c7{I} b8b7{I} b8a7{I} b8c8 2. f7f8{C} c8d7{I} c8c7{I} b8b7 3. d6c7{I} f8g8 b7c6{I} b7b6 4. d6c6{I} g8g5 b6c5{I} b6c6{I} b6b5{I} b6b7 5. d6c7{I} d6d7 b7c6{I} b7b6 6. d7c6{I} d7d6 b6c5{I} b6c6{I} b6b5{I} b6b7 7. d6c6{I} g5c5 b7c6{I} b7b6 8. d7c7{I} d6c6{I} d6d5 b6c5{I} b6b5{I} b6b7 9. d5c6{I} d5d6 b7c6{I} b7b6 10. d6c6{I} d6d5 b6c5{I} b6c6{I} b6b5{I} b6b7 11. d5d6 b7c6{I} b7b6 12. d6c6{I} c5c7 b6c5{I} b6c6{I} b6b5 13. d6c5{I} c7c6 b5c4{I} b5c5{I} b5c6{I} b5b4 14. d6c5{I} d6d5 b4c3{I} b4c4{I} b4c5{I} b4b3 15. d5c4{I} c6d6 b3c3 16. d5c4{I} d5c5 c3d4{I} c3d3{I} c3c4{I} c3d2{I} c3c2 17. c5b4 c2c3{I} c2d3{I} c2b3{I} c2b2 18. b4c3{I} b4a3{I} d6c6 b2c3{I} b2c2{I} b2b3{I} b2c1{I} b2b1 19. b4b3 b1b2{I} b1c2{I} b1c1{I} b1a2{I} b1a1 20. c6c4 a1b2{I} a1b1 21. c4a4 b1b2{I} b1c2{I} b1c1 22. a4d4 c1b2{I} c1c2{I} c1d2{I} c1d1{I} c1b1 23. d4d1{C} 1-0 {White mates}
8. Future Work and Conclusions

In this paper we have described a program which plays a Kriegspiel endgame. We started from a normal chess program and modified it to deal with the uncertainty typical for Kriegspiel playing. In order to evaluate our player, we have played several thousands of games showing that the evaluation function developed is a good basis for further refinements.

We could have implemented a rule-based player based on the procedures reported in Leoncini and Magari (1980) and Boyce (1981). A first problem is that these papers do not prove that their procedures are correct and complete. So, we have no guarantee to obtain a program playing perfectly the KR vs K ending. Moreover, any rule-based solution would have been specialized in KR vs K only. Instead we have adapted our player rather easily to another ending, namely KQ vs K, and now we plan to make similar experiments for other basic endings such as KBBK, KBNK, etc.

References


Williams, J. D. (1950). Kriegspiel rules at RAND. (Unpublished manuscript).

SEARCHING WITH ANALYSIS OF DEPENDENCIES IN A SOLITAIRE CARD GAME

B. Helmstetter, T. Cazenave
Universite Paris 8, laboratoire d’Intelligence Artificielle
2, rue de la Liberte 93526 Saint-Denis Cedex France
{bh, cazenave}@ai.univ-paris8.fr

Abstract We present a new method for taking advantage of the relative independence between parts of a single-player game. We describe an implementation for improving the search in a solitaire card game called Gaps. Considering the basic techniques, we show that a simple variant of Gaps can be solved by a straightforward depth-first search (DFS); turning to variants with a larger search space, we give an approximation of the winning chances using iterative sampling. Our new method was designed to make a complete search; it improves on DFS by grouping several positions in a block, and searching only on the boundaries of the blocks. A block is defined as a product of independent sequences. We describe precisely how to detect interactions between sequences and how to deal with them. The resulting algorithm may run ten times faster than DFS, depending on the degree of independence between the subgames.

Keywords: depth-first search, dependency-based search, block search, Gaps

1. Introduction

In this paper we consider a solitaire card game usually called Gaps, Montana, Rangoon or Blue Moon. We give approximations of winning chances for the game of Gaps and use the domain for testing new ideas. In the field of solitaire card games, we may also mention the game Freecell which has become a test domain in planning (Hoffmann, 2001).

We have reasons to believe that techniques based on heuristics are not very useful in Gaps. However we have been able to improve the search in another way, by proving the independence between moves in different parts of the game and making use of it. A few search techniques with similar concerns exist but they are based on different principles (Allis, 1994; Junghanns and Schaeffer, 2001; Botea, Müller, and Schaeffer, 2002).
This paper is organised as follows. We explain the rules of Gaps in Section 2. We give results of the basic techniques and explain the reasons for our approach in Section 3. We present our method in Section 4 and experimental results in Section 5. Finally, in Section 6 we discuss possibilities for generalization and compare our method to the existing one which is the closest: dependency-based search (Allis, 1994).

After our experiments, it was a surprise to find that a game called Superpuzz, studied by Berliner and more recently by Shintani, is a variant of Gaps. However, at the time this paper was written, we did not know precisely what work has been done on Superpuzz. We have found a short description of Berliner’s (1997) work on the web, and Shintani’s (1999, 2000) work has only been published in Japanese.

2. Rules of the Game

Below we explain the rules of what we call the basic variant (2.1). Then we describe a few other variants (2.2). Finally, we give some basic properties of the game (2.3).

2.1 Basic Variant

The game is usually played with a 52-card deck. The cards are placed in 4 rows of 13 cards each. The 4 Aces are removed, resulting in 4 gaps in the position; then they are placed in a new column at the left in a fixed order (e.g., 1st row Spade, 2nd Heart, 3rd Diamond, 4th Club). The goal is to create ordered sequences of the same suit, from Ace to King, in each row. A move consists in moving a card to a gap, thus moving the gap to where that card was. A gap can be filled only with the successor of the card on the left (that is, the card of the same colour and one higher in value), provided that there is no gap on the left and that the card on the left is not a King, in which case we can place no card in that gap. Figure 1 shows an initial position with only 4 cards per suit, before and after moving the Aces, and the possible moves.

![Figure 1](image-url)  
Figure 1. An initial position with 4x4 cards, before and after moving the Aces. This position can be won.
2.2 Other Variants

The basic variant presented above is probably not the most common. Usually the Aces are not placed in a new column but are definitively removed. Instead it is allowed to place any Two in a gap if it is on the first column. This gives more possibilities than in the basic variant where only one Two can go in each place of the second column (which was the first before moving the Aces). This difference is not a minor one, as it has a strong influence both on the size of the search space and on the probability of a winning game, as we will see. We call this variant the common one. The reason for our choice of the basic variant was to make the rules cleaner; this way only one card can be placed in any gap.

The game is usually played with an additional rule which says that when the player gets stuck, he may remove the cards that are not part of an increasing sequence starting from the first column, and redeal them. Two redeals are allowed. We have not studied the game with this rule. However, as we will see, the probability of winning without this rule but with perfect play is higher than that obtained by human players using this rule.

It is possible to change the number of suits and the number of cards per suit. This influences the size of the search space and the problem’s difficulty. It also has an effect on the degree of independence between subgames, which will be a major concern.

The game that has been studied under the name Superpuzz is what we have called the common variant. There is only one minor difference: the gaps are created by removing the Kings instead of the Aces.

2.3 Properties

In the basic variant, every initial dealing results in a separate game of perfect information. This version has a remarkable property: in any position, the depth of the search graph is bounded; in particular there is no cycle. If we look at a particular card, of value \( v \), there is only \( v - 1 \) places where it could be in the subsequent positions, in addition to its present location: it could be one space on the right of the card of the same suit and of value \( v - 1 \), two spaces on the right of the card of the same suit and of value \( v - 2 \) (which means that we have built a sequence \( v - 2, v - 1, v \) from the current position of the card of value \( v - 2 \) and of the same suit) \ldots, and \( v - 1 \) spaces on the right of the Ace of the same suit. The card cannot go to any of those places twice, so the number of moves of this card is bounded by \( v - 1 \). Therefore the total number of moves with 52 cards is bounded by \( 4 \times (1 + 2 + 3 + \ldots + 12) = 312 \).
3. Basic Techniques

In this section we show results of either a depth-first search or an iterative sampling search applied to Gaps, then we discuss possibilities for improvement. Although the techniques exposed here are basic, the results are about the best we could do without using the method, block search, that we present in the next section.

3.1 Depth-First Search

It turns out that in the basic variant the search space is small enough to allow a complete depth-first search enhanced with a transposition table. Assuming that we stop the search as soon as we find a winning path, the average size of the search space is about 250,000. It seldom goes above 2 million. This is small enough for all positions to be stored in a transposition table. A test on 10,000 initial positions shows that the probability that an initial position can be won is about 24.8%. The length of winning solutions is usually in the range of 90 to 130 moves. All the computations have been made on an Athlon 1600+ with 1GB RAM. The previous search takes about 0.2s per problem.

This is only the beginning of the story though, because the basic variant is far from being the most difficult one, and even in the basic variant the difficulty could be increased by playing with more cards.

3.2 Iterative Sampling

DFS is impractical in variants where the size of the search space is too big. Instead, iterative sampling (Harvey and Ginsberg, 1995) has proved to be surprisingly efficient. This consists in playing completely random moves until a goal is found or the player gets stuck, in which case the search restarts at the beginning. This is repeated until a probe is successful or the maximal number of probes is reached.

We give results of this algorithm both for the basic variant and for the common one (where the Aces are definitely removed and any Two can go in the first column). We consider the common variant because the typical size of its search space is too big to allow a complete search in a reasonable time (this property could also have been obtained by increasing the number of cards). This way we also get a first approximation of the winning chances for the common variant, which are unexpectedly high.

Table 1 shows the results of an experiment on a set of 1000 random initial positions. The set of positions is always the same, except that, for accuracy, experiments with fewer than 1000 probes have been made on 100,000 initial positions. One probe takes about 4.5μs. This amounts to about 450 s for $10^8$
probes when they are unsuccessful; on average 425 s and 164 s for the basic and common variants, respectively.

<table>
<thead>
<tr>
<th>max number of probes</th>
<th>success rate basic variant</th>
<th>success rate common variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.046%</td>
<td>0.041%</td>
</tr>
<tr>
<td>10</td>
<td>0.373%</td>
<td>0.396%</td>
</tr>
<tr>
<td>$10^2$</td>
<td>1.37%</td>
<td>2.96%</td>
</tr>
<tr>
<td>$10^4$</td>
<td>7.1%</td>
<td>26.6%</td>
</tr>
<tr>
<td>$10^5$</td>
<td>9.7%</td>
<td>43.1%</td>
</tr>
<tr>
<td>$10^6$</td>
<td>12.8%</td>
<td>53.0%</td>
</tr>
<tr>
<td>$10^7$</td>
<td>14.5%</td>
<td>60.3%</td>
</tr>
<tr>
<td>$10^8$</td>
<td>16.4%</td>
<td>66.9%</td>
</tr>
</tbody>
</table>

Table 1. Results of iterative sampling.

3.3 Combining a Depth-bounded Search with Iterative Sampling

Iterative sampling can be combined with a depth-bounded complete search. One possibility is to make a breadth-first search until exhaustion of memory resources, and to make one or more random probes at each node of this search. The results are better compared to simple iterative sampling, probably because it ensures a better distribution of the probes in the search space. Furthermore this method will also prove some problems impossible when the search space is searched completely.

We have run a test on 100 random initial positions for the common variant. The breadth-first search was limited by the number of positions that could be stored in memory: this number was set to 5,000,000. One random sample was performed at each node. The program took 144 s per problem in average, and it has found solutions for 88 of the initial positions and proved 4 impossible.

3.4 Comparison with Human Performance

Estimations of the chances of winning for human players are based on various sources from the web and on personal experience. The chances of winning when no redeal is allowed are of about 1%. The exact rule concerning the gaps in the first column apparently has little effect on the difficulty of the game for human players, but we have shown it is important for the computer. The last feature must be compared with 24.8% (basic variant, complete DFS), 66.9% (common variant, iterative sampling) and 88% (common variant, combination of breadth-first search and iterative sampling).

With two redeals allowed, chances of winning for human players are of about 25%, still well below 88%.

It is a particularity of our domain that we have a slight chance of winning by doing random moves. The efficiency of the algorithm is due to its covering a well-distributed part of the search space and its avoiding getting lost in large parts of the tree where it is impossible to win.
3.5 Discussion on Possibilities for Improvement

Iterative sampling is a simple and efficient technique for the game of Gaps; however, in the process of designing a search program for solving Gaps, this took us quite some time to realize. Previously, our attempts for solving Gaps and its variants were based on complex best-first search algorithms. We gradually realized that it was important to try and go to the goal often without caring much for the quality of the moves. At that time our algorithms were about as follows: do some tree search in a best-first way, and during this search, from time to time, launch random samplings. We finally found that the strength of our program was almost entirely due to the random samplings.

At the beginning, we had been working on yet another variant of the game. This variant differs from the common one by the rule that we may move in a gap not only the successor of the card on the left but also the predecessor of the card on the right. We felt that there were much more efficient heuristics in this variant. We did get some successes using heuristics, but even there random sampling alone would do about as well.

In the basic variant the situation is worse. What heuristics do we have? First, there is the number of cards that are already in their final location. This is the only simple heuristic we know about, but unfortunately it gives a poor evaluation of the position, as it often happens that most cards only get in their correct location in the endgame. Then there could be heuristics concerning the mobility of the cards, in the present and in the long term, but this is difficult to estimate.

It is possible that good heuristics could be found. However a comparison with human performance shows that we are not so bad with iterative sampling. One can see that one sample by human players is roughly as successful as 100 random samples. This indicates that human players do not use very efficient heuristics anyway. Even if we could do as well as humans on one sample, considering the time that would be needed for computing heuristics it would probably not be interesting. Because heuristics are weak, any best-first search algorithm would also be of limited use. As an example there is the well-known IDA*; we have experimented with it but did not achieve better results than with a depth-first search.

The next part of this paper takes an orthogonal approach to the heuristic one. Our goal is to go through the search space completely, without even caring whether we find a winning sequence. We want to do it faster than a depth-first search would, by simplifying the search space. Thanks to this, we will be able to determine for sure if there is a solution in some problems where depth-first search is not applicable; for instance in the basic variant when playing with more cards.
4. Block Search

In this section we present a new method that aims at proving the independence between some parts of the problem and taking advantage of it, while keeping the search complete.

From now on the focus will be on the basic variant only, because we will make use of the fact that in any position only one card can be placed in a gap. The common variant does not have this property, and therefore more work would be needed in order to generalize the method to this variant.

4.1 Related Work

Among existing search algorithms, the closest to ours is probably the algorithm dependency-based search by Victor Allis (1994). He applied his method to three domains: the double letter problem, and the search of winning sequences in Qubic and GoMoku (the last two, being 2-person games, were first transformed into single agent games). In fact the starting point of our work was a failure to adapt this algorithm to the game of Gaps. A pseudo-code for the algorithm was given, but a function called NotInConflict was not explicit; we believe that this function was easy to write in the domains where the algorithm had been implemented but would be difficult to write in Gaps, at least not statically.

The goal of our method is also similar to that of Junghanns’ Relevance Cuts for Sokoban (Junghanns and Schaeffer, 2001). He suggested that relevance can be approximated by computing an influence between moves, and then penalizing moves that are not relevant to the previous ones. His work was done in the context of an IDA* search, so in his method moves are never definitely eliminated, they may only get a penalty. The method we have developed handles the problem more precisely.

A more recent work on Sokoban (Botea, Müller, and Schaeffer, 2002) addresses the problem in yet another way, by decomposing the position in rooms and precomputing the graph of states in each room. The major difference with our work is that the states in the subgames are precomputed, and this does not seem possible in Gaps.

4.2 Principle of the Method

We name the four gaps $A$, $B$, $C$, $D$, and break the game into four subgames also named $A$, $B$, $C$, $D$. The moves allowed in one subgame are those that use the corresponding gap. If one plays only in one subgame, one makes a linear sequence of moves. This sequence moves the same gap from place to place until getting stuck, which can happen for any of the following two reasons: either there is another gap on the left, or the card on the left is a King. Whereas the
moves in the same subgame are totally ordered, moves from different subgames are often independent. We want to take advantage of this relative independence between the subgames.

Let a block be defined by its starting position, and in each subgame $X$ a sequence $S_X$, possibly empty, from the starting position, so that the four sequences are independent of each other. A block represents a set of positions: all positions that can be reached from the starting position of the block with the moves of the sequences $S_X$ in any order. It helps to see a block geometrically as embedded in a four dimensional space. If $l_X$ is the length of sequence $S_X$, the block represents $l_A \times l_B \times l_C \times l_D$ positions. Finally, we call $F_X$ the face of the block that consists in the positions of the block where all the moves of sequences $S_X$ have been made.

The main idea of the algorithm is: instead of searching one position at a time, we search one block at a time. Instead of recursively searching the immediate children of a position, we construct new blocks at the boundary of a block and recursively search them.

We want to construct blocks of the biggest possible size, so before building blocks on the boundary, we try to extend them as much as possible in the four subgames. Figure 2 shows a pseudo-code for the algorithm.

```c
void search(block) {
  for each subgame $X$
    extend block in the subgame $X$, as long as all the sequences keep being independent;
  for each subgame $X$
    build new blocks near the face $F_X$ of the block, such that any move we can do from $F_X$ goes to one of those blocks, and search them recursively;
    test for a winning position in the block;
}
```

*Figure 2.* Pseudo-code for block search.

We still have to show how to extend blocks and construct new blocks at the boundaries. Besides, the pseudo-code does not include a transposition table, and this will lead to some problems to be addressed in Subsection 4.7.

### 4.3 Study of the Basic Interaction

We study in detail the case of a single interaction between two sequences. Figure 3 shows the useful part of the position and a diagram which synthesizes the relation between the two sequences. For simplicity all cards are of the same suit. We assume that both sequences begin a few moves before the interaction and end a few moves after, although the moves that are not critical have not been
drawn. The dotted arrow in the right diagram indicates the action of sequence $S_B$ on sequence $S_A$. Assume we are just after move $a_1$ in $S_A$. If $b_1$ has not yet been made, move $a_2$ can be made in subgame $A$; if move $b_2$ has already been made, move $a'_2$ can be made in subgame $A$; if move $b_1$ has been made but not move $b_2$, no move can be made in subgame $A$.

Figure 3. A basic interaction.

Figure 4 shows a representation in the plane of the search space, where the positions lie at the intersections of the lines. Imagine there was no interaction between the two sequences; we would have a big square with the entire sequences $A$ and $B$ on the edges. The effect of the interaction is to cut this square along a line from the point $p$ to the right side (the double line in the figure), and to stick another part along the cut, which corresponds to sequence $S_A$ taking the bifurcation. The position at $p$ is particular: it is the position where the two gaps are adjacent, so that no move can be made in subgame $A$. This point corresponds to the dotted arrow in Figure 3. We say that there is a bifurcation of sequence $S_A$, caused by an action of sequence $S_B$. One must imagine that there are two other dimensions corresponding to the subgames $C$ and $D$; if the sequences in these subgames do not introduce new interactions, the complete search space will be a simple product of the graph in Figure 3 with the sequences $S_C$ and $S_D$.

We are looking for ways to partition the search space into blocks. There are several ways to do it; Figure 5 shows the two ways we will use. They deal with the two possible shapes of the first block. Whether we get in the first or in the second depends on the order in which we have extended the first block: first in subgame $A$ or $B$. Subsection 4.6 will explain precisely how to detect interactions when extending blocks and how to build new blocks at the boundary. For the moment, we note that in the first possibility blocks 2 and 3
are children of block 1, whereas in the second only blocks 2 and 4 are children of block 1, and block 3 is a child of block 2.

Figure 5. Two ways to decompose the search space into blocks.

4.4 Why the Basic Interaction is the Only One to Consider

We have shown how to deal with the basic interaction; it turns out that it is the only one we have to consider. Let $S_X$ and $S_Y$ be two sequences in the subgames $X$ and $Y$. Let us enumerate all the ways that a move $y$ of sequence $S_Y$ could be influencing a move $x$ of sequence $S_X$. Move $x$ consists in taking the card $c$ from place $p_1$ to place $p_2$. The prerequisites for this move are:

1. there is a gap at $p_2$,
2. the card $c$ is in $p_1$,
3. the card at the left of $p_2$, $c_L$, is the predecessor of $c$.

Those preconditions are verified if we make only moves from sequence $S_X$ up to $x$, but they could be destroyed by moves of sequence $S_Y$. Assume we have already established the independence of the sequences $S_X$ and $S_Y$ up to the moves $x$ and $y$; then precondition 1 is automatically satisfied as soon as we have executed all the moves of sequence $S_X$ up to $x$ and whatever we have done in the sequence $S_Y$ up to $y$, since it is an effect of the beginning of sequence $S_X$ to put a gap in $p_1$.

Precondition 2 is automatically satisfied too. This comes from the fact that the card $c$ can only go to the right of $c_L$, wherever this card be. If this card was moved by sequence $S_Y$, then there would already be an interaction because of precondition 3. If move $y$ moved card $c$ to the right of $c_L$ and this card was still at the left of $p_2$, the trajectories of the gaps $X$ and $Y$ in the sequences $S_X$ and $S_Y$ up to $x$ and $y$ would both pass through $p_2$, which again would imply that they are dependent.

Therefore only precondition 3 remains to consider, which produces an interaction of the kind already analysed.
4.5  Using a Trace to Speed Up the Discovery of the Interactions

The trace of the sequences $S_X$ is an array of the same size as the position (4x13). It contains information for each place $p$ indicating whether and at which move the trajectory of the gap for any sequence is passing through $p$. The trace is maintained incrementally as we build new blocks.

Assume we make a new move $m$, which moves a card from $p_1$ to $p_2$. We want to know if it produces new interactions with the sequences already built. A look in the trace at the place just on the left of $p_2$ shows if any sequence has an effect on move $m$. A look in the trace at the place just on the right of $p_1$ shows if move $m$ has an effect on any sequence. This way the search for an interaction is done very quickly.

The method of doing a local search and storing the set of properties of the position on which the result relies has already been used by other people in different domains: in Go, with the goal of incrementally updating local results (Bouzy, 1997); in Generalized Threats Search (Cazenave, 2002) which is a 2-players selective search algorithm that relies on a trace to find a set of relevant moves; in the algorithm H-search used in the hex program HEXY (Anshelevich, 2002) with a bottom-up approach, building increasingly complex virtual connections.

4.6  Building and Extending Blocks

The procedure for building blocks at the boundary is tricky, because we have to take into account all the interactions that might occur. Although there is only one kind of interaction that needs to be considered, it can come in the two different configurations shown in Figure 5, and we must be prepared that several configurations occur at a time. Figure 6 represents a search space in two dimensions that gives an idea of the kind of situations we have to deal with.

We are on face $F_B$ of block $b$. We try to find out what moves can be made in subgame $B$ depending on the exact location on $F_B$, and if the moves have an action on the other sequences. This situation occurs twice in the program: first when we are trying to extend the block in subgame $B$ (which can be done as long as there is no interaction), second when we are building new blocks near face $F_B$ (generally because we have already found an interaction). We must answer the following questions in this order.

1. Is there an action of any other sequence of the block that will cause a bifurcation on $S_B$? This is the case if and only if the trajectory of the gap for any other sequence is passing through the place at the left of gap $B$. This can be decided quickly by looking at the trace. An example is interaction 1 in Figure 6.
2 Now we know that the move to be made in subgame $B$ does not depend on the location on face $F_B$, or we have already restricted ourselves to an area where it is the case. Is a move possible at all? There cannot be another gap on the left of gap $B$ because of the previous step, so the question is whether the card on the left is a King (in this case the block cannot be extended, and no block will be constructed on this part of the boundary).

3 Now we know we can do a move; this move takes a card from a place $p$ and moves it in gap $B$. Is there an action of this move on the other sequences? This is the case if and only if the trajectory of the gap for any other sequence is passing through the place at the right of $p$. This too can be decided quickly by looking at the trace. An example is interaction 2 in Figure 6.

When building new blocks at the boundary, one must go through these three steps. In steps 1 and 3 we may have to break face $F_B$ into two parts (in some degenerated cases there may be one or zero part) and apply the following steps to each. After step 3 we have isolated a part $G$ of the face $F_X$. We know that we can make a move $m$ anywhere on $G$ and that this move has no effect on the other sequences. We then create a new block by doing move $m$ from $G$, and search it recursively.

A block $c$ that has just been built on face $F_X$ of a block $b$ has no depth in subgame $X$. When we try to extend block $c$ in all the subgames, it is generally successful for subgame $S_X$; on the contrary, it is generally not successful in the other subgames because the reasons why block $b$ had been stopped in those subgames often stand for block $c$. 

*Figure 6. Several interactions on one face of a block.*
4.7 Adding a Transposition Table

The decomposition in blocks already handles all the transpositions within the blocks; this is good but insufficient. In order for the algorithm to be efficient, it is almost compulsory to have a global transposition table. However, when we construct a new block or when we extend one, there could be common positions anywhere in this block and in another already built one.

Now we do not want to go to all the positions in every block and mark them in the transposition table, because the advantages of our method would disappear. We have to look for a compromise: we could mark only a well-chosen part of each block and hope it will be sufficient to detect most transpositions.

We define the number of positions of a block search as the sum, for each block, of the number of positions contained in this block. Let $R$ be the ratio between the number of positions of a block search and the number of positions of a depth-first search. Ideally, if the transpositions table is large enough to contain all the positions of the search space and if the blocks are mutually disjoint, then $R = 1$. If the blocks are not mutually disjoint, then $R$ will be larger; we need to control how much larger it will be.

A first possibility is to mark only the starting position of each block. An experiment on 100 random positions for the basic variant has shown that the ratio $R$ is about 3.95, which is too much.

A second possibility is to mark only the positions that can be reached from the initial position of the block by making moves in only one of the four sequences of the block simultaneously. Geometrically, those are the points located on the four edges of the block starting from the initial position. The ratio $R$ drops down to 1.33, which is acceptable although a better compromise probably exists.

5. Experimental Results

The method was designed to be complete; we have verified experimentally that it is indeed the case. This is a sign that we have correctly analysed all the possible interactions that can occur at the boundaries of the blocks. The method for verifying the completeness was the following: from an initial position, first run a complete depth-first search and store all the positions of the search space; then run a block search and verify that all the positions of the search space lie in at least one of the blocks. This verification has been done for 1000 initial positions.

Table 2 shows statistics about an experiment on 1000 random initial positions for the basic variant. There is a difference in time and number of positions compared to Subsection 3.1 because the search is not stopped when we find a winning position. Also the transposition table is not implemented in the same way: before it could grow as needed, now we use a hashtable of fixed size as
is usual in game programming (Breuker, 1998). The hashtable has 64 million entries.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>avg. number of positions</td>
<td>502,000</td>
</tr>
<tr>
<td>for DFS</td>
<td></td>
</tr>
<tr>
<td>avg. number of blocks</td>
<td>36,200</td>
</tr>
<tr>
<td>avg. size of blocks</td>
<td>18.6</td>
</tr>
<tr>
<td>$R$</td>
<td>1.34</td>
</tr>
<tr>
<td>avg. time for DFS</td>
<td>2.28s</td>
</tr>
<tr>
<td>avg. time for block search</td>
<td>2.04s</td>
</tr>
</tbody>
</table>

*Table 2.* Basic variant (4 suits, 13 cards/suit).

The average size of the blocks is 18.6, so one node of block search does as much work as 18.6 nodes of DFS in average. As we have already mentioned, the total number of positions in all the blocks is larger than the number of positions searched by DFS, by about 34%. The final result in speed is a gain of 11% for blocks search. In the present case however, the difference in time is not very significant of the performance of block search because, for both algorithms, much time is spent reinitializing the large transposition table between problems.

We do not see the power of block search yet. Higher gains in speed can be obtained in variants with a larger search space, and with a higher degree of independence between the subgames. This can be achieved by increasing the number of cards. We therefore turn to 6 suits and 13 cards per suit. This increases both the number of cards and the number of subgames.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>number of positions for</td>
<td>289 × 10^6</td>
</tr>
<tr>
<td>DFS</td>
<td></td>
</tr>
<tr>
<td>number of blocks</td>
<td>5.00 × 10^6</td>
</tr>
<tr>
<td>size of blocks</td>
<td>59.7</td>
</tr>
<tr>
<td>$R$</td>
<td>1.03</td>
</tr>
<tr>
<td>time for DFS</td>
<td>437s</td>
</tr>
<tr>
<td>time for block search</td>
<td>44s</td>
</tr>
</tbody>
</table>

*Table 3.* 6 suits, 13 cards/suit, the hashtable has 64 million entries.

It is difficult to give average statistics because the size of the problems vary a great deal, some being too big for DFS and a few even for block search. We have made 15 experiments with initial positions that could be completely searched both with blocks search and DFS. In Table 3 we show detailed statistics for one of them, which is typical. We also show in Figure 7 that the gain in speed is correlated to the size of the search space. This

![Figure 7](image-url)  
*Figure 7.* Gain in speed of block search over DFS, for 15 initial positions.
Searching with Analysis of Dependencies in a Solitaire Card Game

is promising. Our algorithm clearly has an advantage over a depth-first search because it can build large blocks, and this advantage would grow larger if the number of suits and/or cards per suit was further increased.

<table>
<thead>
<tr>
<th>number of positions for DFS</th>
<th>$1550 \times 10^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of blocks</td>
<td>$5.88 \times 10^6$</td>
</tr>
<tr>
<td>size of blocks</td>
<td>61.1</td>
</tr>
<tr>
<td>$R$</td>
<td>0.23</td>
</tr>
<tr>
<td>time for DFS</td>
<td>1730s</td>
</tr>
<tr>
<td>time for block search</td>
<td>46s</td>
</tr>
</tbody>
</table>

Table 4. 6 suits, 13 cards/suit, the hashtable has 8 million entries.

In the experiment of Table 3, the ratio $R$ has dropped down from 1.34 to 1.03. This is due to collisions in the transposition table. This phenomenon is amplified if we decrease the size of the transposition table: Table 4 shows statistics about the same problem but with a hashtable that can only contain 8M positions. This has a dramatic effect on DFS, but almost no effect on our algorithm. Because of collisions, the ratio $R$ is even less than 1! So, now the situation is reversed: it is our algorithm that makes a better use of the transposition table.

6. Perspectives

We conclude the paper by providing two perspectives. In Subsection 6.1 possibilities for generalization are given. In Subsection 6.2 dependency-based search is compared to block search.

6.1 Possibilities for Generalization

The general idea of the method does not rely much on the domain of Gaps. Our notion of a block can in principle find equivalents in many domains, provided that we generalize it a little. Until now we have worked with blocks that are products of independent sequences; as a first generalization, we should define blocks as products of independent graphs. In most domains there will be parts of the problem that will be, at least locally, relatively independent.

To apply the method, we must define what a subgame is, by stating which moves belong to which subgame, and we must analyse precisely all kind of interactions that could occur between them. This analysis is difficult and is domain-dependent, but then the rest is similar to what we did in Gaps: build and extend subgraphs in each subgame only as long as they keep being independent. The product of those graphs gives a block. Then we build new blocks at the boundary of this block and search them recursively.

Therefore we claim that the idea of decomposing the search space in blocks is a natural way to simplify the search space and may be applicable to other domains. Furthermore, the method could be much more powerful in domains with more independence between subproblems, leading to the construction of much larger blocks.
6.2 Comparison between Dependency-based Search and Block Search

As a first application domain to dependency-based search, Allis (1994) considered the double letter problem. In this domain, a state consists in a word on the set of 5 letters \( \{a, b, c, d, e\} \). Any double occurrence of a letter can be replaced by a single instance of its alphabetical predecessor or successor. The alphabet is cyclic so \( ee \) can be replaced either by \( d \) or \( a \), \( aa \) by either \( e \) or \( b \). The winning states are the one-letter words. A detailed application of the method had been shown by Allis for the initial state \( aaccadd \). A solution exists (the letters that change have been capitalized):

\[
aaccadd \rightarrow Bccadd \rightarrow bBadd \rightarrow Aadd \rightarrow Edd \rightarrow eE \rightarrow A
\]

We are going to compare the way this instance is solved by dependency-based search (according to Allis) and a way it could be solved by a block search algorithm. Dependency-based search runs with a succession of dependency stages and combination stages. After one dependency stage and one combination stage, he gets the graph in Figure 8: he considers the 6 moves possible at the root and finds that two can be combined together. The same situation can be represented with blocks (Figure 9): we have 3 independent subgames corresponding to the letters at the positions 1, 2, 3, 4 and 6, 7, respectively. In each of those subgames two moves can be made from the initial position. Therefore the set of positions reachable with these moves can be represented with a cube, the initial position \( aaccadd \) being in the centre. We then find an interaction at one of the edges of the cube: the two “B” that have been created allow to move in a new subgame and therefore a new block can be constructed.

![Figure 8. Dependency-based search, beginning.](image)

The rest of the search continues similarly with dependency-based search (Figure 10) and block search (Figure 11). At least in this example we are really doing the same thing with different representations.

This goes to show that both methods have similarities. However, there are some differences that cannot easily be seen on the last example. First we do not see all the power of block search here: comparatively to dependency-based search, we believe it can deal with interactions of a more complicated nature (as in Gaps where we could not apply dependency-based search). Probably we
Searching with Analysis of Dependencies in a Solitaire Card Game

Figure 9. Block search, beginning.

Figure 10. Dependency-based search, complete.

Figure 11. Block search, complete.

do not see all the power of dependency-based search either. For instance and in contrary to block search, it is not necessary to provide an explicit decomposition in subgames to apply dependency-based search.

7. Conclusion

We have presented several search algorithms that take advantage of the particularities of the game of Gaps. Our work has resulted in a method, block search, which may be applicable in other domains.

We have shown that iterative sampling produces good results, either for the basic variant or the common variant. Conversely, we have shown that the use of heuristics is not so promising. Therefore we could deal with only one problem in isolation: exploiting the independence between parts of the game. Existing methods that deal with this problem were either not applicable to the domain of Gaps or were not as precise as ours.
Block search is a method to take advantage of a decomposition in subgames when there are interactions between those subgames, while keeping the search complete. It implies analysing theoretically all types of interactions that can occur: how to detect them, how to deal with them by building new blocks at the boundary of the current block. Although this analysis relies on domain-dependent knowledge, the general idea of the method does not. Experimental results have shown that large gains in speed over a depth-first search can be expected, depending on the average size of the blocks we are able to build. Specifically, the method can be used to solve positions of the basic variant of Gaps with more cards. Because the method simplifies the search space, it also makes better use of a transposition table.

References


SOLVING THE OSHI–ZUMO GAME

M. Buro
University of Alberta, Edmonton, AB T6G 2E8, Canada
mburo@cs.ualberta.ca, http://www.cs.ualberta.ca/~mburo/

Abstract Kotani (2002) determined the part of the state space of the Japanese Oshi-Zumo game in which pure strategies suffice to win. This paper completes the analysis by computing and discussing a Nash-optimal mixed strategy for this game.

Keywords: Nash-optimal strategy, Oshi-Zumo, two-player game

1. Introduction

In this article the Japanese game Oshi-Zumo is analyzed. Moves in this game consist of simultaneous actions by two players who otherwise have complete information about the current game state. In general, such games can be represented by a collection of payoff matrix pairs whose entries define the expected amount paid to the players in case the respective action pair was chosen. It is well known that not knowing the opponent’s action already makes it necessary to consider mixed strategies and that so-called Nash-optimal mixed strategies exist for any matrix game (Nash, 1950). A simple example is the Rock-Paper-Scissors game in which Rock beats Scissors, Scissors beats Paper, and Paper in turn beats Rock. The Nash-optimal strategy picks each of the actions with probability $\frac{1}{3}$.

In what follows, we first introduce the Oshi-Zumo game. It is more complex than Rock-Paper-Scissors, but considerably simpler than other popular incomplete information games such as Poker and Bridge. In fact, we will show how to compute a Nash-optimal strategy within seconds on ordinary PC hardware. We then highlight interesting properties of a Nash strategy and conclude the paper by discussing how the optimal player performs against reasonable, but sub-optimal strategies.

2. The Game

Oshi-Zumo – meaning “the pushing sumo (wrestler)” – is played by two players who both start off with $N$ coins. At the beginning of a game, a sumo
362

M. Buro

[50,4,*]-Oshi-Zumo starting position – code (50,50,0)

Position after move (4,2) – code (46,48,1)

Figure 1. Oshi-Zumo positions and their triple representation.

wrestler is positioned at the center of a one-dimensional playing field which consists of $2K+1$ locations (see Figure 1). Moves are played by secretly choosing a number of coins less or equal to the amount currently available to the respective player, but at least $M$. The bids are then revealed and the highest bidder pushes the wrestler one location towards the opponent’s side. If the bids are equal, the wrestler does not move. Both bids are deducted and the game proceeds until the money runs out or the wrestler is pushed off the playing field. The final position of the wrestler determines the winner: if he is located at the center, the game result is a draw. Otherwise, the player in whose half the wrestler is located loses the game. We call this parameterized game an $[N,K,M]$-Oshi-Zumo game. In this paper we only consider the minimal bids $M = 0$ and $M = 1$ and declare a game over if both bids are 0. As before, the winner is determined by the current wrestler position.

3. Computing a Nash-Optimal Strategy

Certain Oshi-Zumo positions possess pure winning strategies. For example, all positions in which the opponent has no money left and the wrestler position is sufficiently advanced can be won by simply bidding one coin for the remainder of the game. Kotani (2002) determined all such positions for the standard $[50,3,1]$-Oshi-Zumo game. The following list specifies some more interesting $[50,4,0]$-positions that can be won by the first player with a pure strategy:

$$(n,n,1) : 1 \leq n \leq 11 \text{ [bid 1]} \quad (n,n+1,2) : 1 \leq n \leq 12 \text{ [bid 1]}$$

$$(50,n,-4) : 1 \leq n \leq 16 \text{ [bid n]} \quad (49,n,-4) : 1 \leq n \leq 16 \text{ [bid n]}$$

All such positions can be computed by dynamic programming for small values of $N$ and $K$ because the size of the state space is only a polynomial $(N+1)^2 \times (2K+3)$ in the parameters. First, we compute the payoffs $P_1$ for both players at the boundary positions:

$$P_1(0,0,k) = -P_2(0,0,k) = \text{sign}(k), \text{ for } -K \leq k \leq K$$

$$P_1(n,m,\pm(K+1)) = -P_2(n,m,\pm(K+1)) = \pm 1, \text{ for } 0 \leq n,m \leq N$$
maximize $Z$ such that
for all $M \leq j \leq n_2$: $Z \leq \sum_{i=M}^{n_1} A_{i,j} x_i,$
for all $M \leq i \leq n_1$: $x_i \geq 0$, and
$\sum_{i=M}^{n_1} x_i = 1$

minimize $Z$ such that
for all $M \leq i \leq n_1$: $Z \geq \sum_{j=M}^{n_2} A_{i,j} y_j,$
for all $M \leq j \leq n_2$: $y_j \geq 0$, and
$\sum_{j=M}^{n_2} y_j = 1$

Figure 2. Linear programs (LPs) for determining Nash-optimal mixed strategies.

Then we search for positions with pure winning or drawing strategies, or ones that lose for sure no matter what. A position is won for player $A$ if there exists an action such that for all actions of the opponent the expected payoff for $A$ is 1. Declaring a position drawn or lost requires that all successor position values are known. We repeat this process until we do not find any new position values.

A Nash-optimal strategy can be computed similarly. Starting again with assigning values to the boundary positions, we iterate through all positions with unknown expected payoff until we find one for which all successor values have been established. At this time we make use of the fact that optimal strategies $\{(i, x_i) \mid M \leq i \leq n_1\}$ and $\{(j, y_j) \mid M \leq j \leq n_2\}$ for players $\text{MAX}$ and $\text{MIN}$ can be found by solving two linear programs (see Figure 2). $\text{MAX}$ has move choices $M, \ldots, n_1$ and $\text{MIN}$ has actions $M, \ldots, n_2$. $x_i$ and $y_j$ denote the respective action probabilities. Matrix element $A_{i,j}$ defines the payment for $\text{MAX}$ if action pair $(i, j)$ is chosen. Because Oshi-Zumo is a zero-sum game, $\text{MIN}$ receives the negated amount. $Z$ denotes the expected payoff for $\text{MAX}$. This procedure eventually halts and computes the expected payoffs and mixed strategies for all positions.

We decided to not only create a table containing expected payoffs – which would be sufficient for computing values for all positions – but also to store the move distributions to speed up later game play and move analyses. Only one distribution needs to be computed and stored for each position because the move distribution for the second player in position $(n, m, k)$ is identical to that of the first player at $(m, n, -k)$.

4. Implementation Issues

In our first implementation we adopted Michel Berkelaar’s open-source software package LPSOLVE. Unfortunately, the solver ran into numerical problems which caused it to either give up on instances or report incorrect solutions. Implementing efficient LP solvers is by no means easy. In order to overcome the numerical problems we decided to replace floating-point by rational arithmetic in LPSOLVE – which turned out to be more complicated than expected. Finally,
we took the simpler LP solver code from Press et al. (1992) and combined it with GMP – the GNU arbitrary precision arithmetic library – by replacing the float/double data types by GMP’s rational number C++ class. Solving LPs using rational arithmetic takes much longer than using floating-point values, even if the denominators are bounded. In order to speed up the Oshi-Zumo solver we therefore implemented a two-phase approach: whenever the fast LP solver reported problems or produced inconsistent results, we would start the slow solver based on rational arithmetic. We bounded denominators by $10^8$ and normalized rational numbers whenever this limit was exceeded. Test runs on Oshi-Zumo games manageable by the floating-point based solver indicated that the results obtained by rational arithmetic only differed by a negligible amount. On a notebook PC with a 1-GHz Pentium-III CPU, solving the standard $[50, 3, 1]$ game takes just 12 seconds. The C++ source code can be downloaded from http://www.cs.ualberta.ca/~mburo/sumo.tgz.

5. A Nash-Optimal Oshi-Zumo Strategy

In what follows we concentrate on the $[50, 3, 0]$ and $[50, 3, 1]$ versions of the game and highlight interesting properties of their respective Nash-optimal strategies. We start by looking at the move distributions for the starting position:

\[ M = 0 \quad \text{position=(50, 50, 0)} \quad \text{value=0.0} \]

<table>
<thead>
<tr>
<th>bids</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob</td>
<td>.083</td>
<td>.077</td>
<td>.088</td>
<td>.083</td>
<td>.092</td>
<td>.088</td>
<td>.097</td>
<td>.092</td>
<td>.099</td>
<td>.094</td>
<td>.101</td>
</tr>
</tbody>
</table>

\[ M = 1 \quad \text{position=(50, 50, 0)} \quad \text{value=0.0} \]

<table>
<thead>
<tr>
<th>bids</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob</td>
<td>.139</td>
<td>.053</td>
<td>.146</td>
<td>.060</td>
<td>.152</td>
<td>.067</td>
<td>.156</td>
<td>.068</td>
<td>.156</td>
</tr>
</tbody>
</table>

Apparent is an “odd-even” effect in which higher and lower bid probabilities alternate. This probability pattern occurs in many positions. Why it occurs is an open question.

The smallest positions with randomization requirement are $(5, 2, -3)$ for $M = 0$ and $(6, 3, -3)$ for $M = 1$. The move distributions are as follows:

\[ M = 0 \quad \text{position=(5, 2, -3)} \quad \text{value} = -0.5 \]

<table>
<thead>
<tr>
<th>bid$_1$:</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob:</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

\[ M = 1 \quad \text{position=(6, 3, -3)} \quad \text{value} = -0.5 \]

<table>
<thead>
<tr>
<th>bid$_1$:</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob:</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bid$_2$:</th>
<th>0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob:</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bid$_2$:</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>prob:</td>
<td>.5</td>
<td>.5</td>
</tr>
</tbody>
</table>

In 5,271 cases of the 23,409 possible $[50, 3, 0]$-positions, and in 4,057 cases for $M = 1$, more than one move has to be considered. To illustrate how complex the move decision can be, we present two positions with a high number of holes in the move distribution:
Solving the Oshi-Zumo Game

Given such complex distributions, the question arises how well human players can play Oshi-Zumo.

6. How Good is Optimal?

Playing any mixed or pure strategy against a Nash-optimal player results in an expected payoff no better than the expected value $E$ of a game between two Nash players. In contrast, the expected value of any pure strategy that picks actions from the set an optimal strategy considers, is exactly $E$ when playing against the Nash-optimal player. This follows from the fact that all actions with non-zero probability have the same expected value. Therefore, the Nash-optimal solution is far from optimal with respect to exploiting simple (pure) strategies, such as playing Rock all the time in a sequence of Rock-Paper-Scissors games. In Rock-Paper-Scissors the Nash strategy cannot win anything against any other strategy in the long run. However, in more complex games – such as Oshi-Zumo or Poker – it can, because not all actions have non-zero probability in all situations.

A player who just memorizes one move from a Nash-optimal strategy for each position does not lose money against a Nash-player in the long run. How much does a player lose who occasionally plays moves not played by a Nash-player and how well do simple hand-crafted strategies play? To answer these questions we wrote a program that played a large number of games between a Nash-optimal strategy and several simple move selection algorithms. Figure 3 presents the tournament results. As expected, the completely random player loses almost every game. The player that randomly chooses bids in the interval formed by the minimal and maximum Nash bid performs much better and loses only about 0.035 units per game for $M = 0$ and 0.01 for $M = 1$. Simply choosing moves in a small fixed interval also leads to good results and shows how easy it is to look good against a Nash player. Also some fairly simple pure strategies perform surprisingly well.

A more interesting question is therefore how to adapt to players and exploit their weaknesses while minimizing the risk of being exploited. We think that
\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{M = 0} & \textbf{M = 1} \\
\hline
random 0..# & random 1..# \\
\hline
random Nash range & random Nash range \\
\hline
random 1..min(6,#) & random min(2,#)..min(6,#) \\
\hline
random 1..min(5,#) & random min(2,#)..min(5,#) \\
\hline
random 1..min(4,#) & random min(2,#)..min(4,#) \\
\hline
random 1..min(3,#) & random min(2,#)..min(3,#) \\
\hline
random 1..min(2,#) & \\
\hline
\text{1} & if \text{#} \geq 2 \\
\hline
\text{if \#} \geq 2 \text{ 2 else 1} & if \text{#} \geq 3 \\
\hline
\end{tabular}
\caption{The average payoff of various simple move-selection algorithms playing 200,000 \([50, 3, M]\)-games against a Nash-optimal strategy. \# denotes the current number of coins left for the heuristic player.}
\end{table}

using games simpler than say Poker but harder than Rock-Paper-Scissors as test domains can shed light into this interesting problem, which appears to be the last remaining hurdle on the way to Poker programs stronger than human players (Billings et al., 2003). Oshi-Zumo is a suitable candidate because its Nash-optimal strategy is non-trivial, but can be computed quickly.

Acknowledgement

Thanks go to Darse Billings for helpful discussions clarifying questions on Nash-optimal strategies.

References


NEW GAMES RELATED TO OLD AND NEW SEQUENCES

A.S. Fraenkel

Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel

fraenkel@wisdom.weizmann.ac.il, http://www.wisdom.weizmann.ac.il/~fraenkel/

Abstract We define an infinite class of 2-pile subtraction games, where the amount that can be subtracted from both piles simultaneously, is a function \( f \) of the size of the piles. Wythoff’s game is a special case. For each game, the 2nd player winning positions are a pair of complementary sequences, some of which are related to well-known sequences, but most are new. The main result is a theorem giving necessary and sufficient conditions on \( f \) so that the sequences are 2nd player winning positions. Sample games are presented, strategy complexity questions are discussed, and possible further studies are indicated.

Keywords: 2-pile subtraction games, complexity of games, integer sequences

1. Introduction

Dominican International Forwarding is the finest international transportation company, according to its web site (in Spanish) at http://www.dif.com.do. (An optional Google rendition confirms once again that mechanical translation is still in its infancy.) What is the connection of DIF to games?

While pondering this question, let us introduce our first game:

**G\(_1\) from dif.com**

Given two piles of tokens \((x, y)\) of sizes \(x, y\), with \(0 \leq x \leq y < \infty\). Two players alternate removing tokens from the piles.

(a) Remove any positive number of tokens from a single pile, possibly the entire pile.

(b) Remove a positive number of tokens from each pile, say \(k, \ell\), so that \(|k - \ell|\) is not too large with respect to the position \((x_1, y_1)\) moved to from \((x_0, y_0)\), namely, \(|k - \ell| < x_1 + 1 \ (x_1 \leq y_1)\).

The player making the move after which both piles are empty (a *leaf* of the game), wins; the opponent loses. Thus, \((11, 15) \rightarrow (3, 4)\) or to \((2, 4)\) are legal
Table 1. The first few $P$-positions for $G_1$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>11</td>
<td>17</td>
<td>25</td>
<td>34</td>
<td>44</td>
<td>55</td>
<td>68</td>
<td>82</td>
<td>97</td>
<td>113</td>
<td>130</td>
<td>149</td>
<td>169</td>
<td>190</td>
</tr>
</tbody>
</table>

Table 1. The first few $P$-positions for $G_1$.

moves, but $(11, 15) \to (2, 3)$ or to $(0, 3)$ are not. The position $(0, 0)$ is the only leaf of this and our following games.

For any acyclic combinatorial game without ties, such as $G_1$, a position $u = (x, y)$ is labeled $N$ (Next player win) if the player moving from $u$ can win; otherwise it is a $P$-position (Previous player win). Denote by $\mathcal{P}$ the set of all $P$-positions, by $\mathcal{N}$ the set of all $N$-positions, and by $F(u)$ the set of all (direct) followers or options of $u$. It is easy to see that for any acyclic game,

$$u \in \mathcal{P} \quad \text{if and only if} \quad F(u) \subseteq \mathcal{N}, \quad (1)$$

$$u \in \mathcal{N} \quad \text{if and only if} \quad F(u) \cap \mathcal{P} \neq \emptyset. \quad (2)$$

Indeed, player I, beginning from an $N$-position, will move to a $P$-position, which exists by (2), and player II has no choice but to go to an $N$-position, by (1). Since the game is finite and acyclic, player I will eventually win by moving to a leaf, which is clearly a $P$-position.

Let $S \subset \mathbb{Z}_{\geq 0}$, $S \neq \mathbb{Z}_{\geq 0}$, and $\overline{S} = \mathbb{Z}_{\geq 0} \setminus S$. The minimum excluded value of $S$ is

$$\mex S = \min \overline{S} = \text{least nonnegative integer not in } S.$$  

Note that $\mex$ of the empty set is 0.

Table 1 portrays the first few $P$-positions $(A_n, B_n)$ of $G_1$. The reader is encouraged to verify that the first few entries of the table are indeed $P$-positions of the game. For a technical reason we put $B_{-1} = -1$. In Section 4 we prove, as a simple corollary to a considerably more general result,

**Theorem 1.** For $G_1$, $\mathcal{P} = \bigcup_{i=0}^{\infty} (A_i, B_i)$, where, for all $n \in \mathbb{Z}_{\geq 0}$,

$$A_n = \mex \{ A_i, B_i : 0 \leq i < n \}, \quad (3)$$

$$B_n = B_{n-1} + A_n + 1. \quad (4)$$

The game $G_1$ is a special case of the following new family of combinatorial games defined on two piles of finitely many tokens, with two types of moves: a move of type (a), and a more general move of type (b), namely, $|k - \ell|$ depends
New Games Related to Old and New Sequences

Table 2. The first few P-positions for $G_2$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>24</td>
<td>26</td>
<td>30</td>
<td>32</td>
<td>34</td>
<td>38</td>
<td>40</td>
<td>42</td>
<td>46</td>
</tr>
</tbody>
</table>

Table 3. The first few P-positions for $G_3$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>11</td>
<td>20</td>
<td>38</td>
<td>71</td>
<td>136</td>
<td>265</td>
<td>523</td>
<td>1036</td>
<td>2061</td>
<td>4110</td>
<td>8207</td>
<td>16400</td>
<td>32785</td>
<td>65554</td>
</tr>
</tbody>
</table>

on the present and next position. Denote the present position by $(x_0, y_0)$ and the position moved to by $(x_1, y_1)$. We then require,

$$|(y_0 - y_1) - (x_0 - x_1)| = |(y_0 - x_0) - (y_1 - x_1)| < f(x_1, y_1, x_0), \quad (5)$$

where $f$ is a real constraint function depending on $x_1, y_1, x_0$. If also $(y_0 - x_0) \geq (y_1 - x_1)$, then the requirement becomes $y_0 < f(x_1, y_1, x_0) + y_1 - x_1 + x_0$.

The type (b) move defined for $G_1$ is the special case $f = x_1 + 1$. Here are descriptions of two additional games.

**$G_2$ from even.com**

Same as $G_1$, except that in (b), $|k - \ell| < x_1 + 1$ is replaced by $|k - \ell| < x_0 - x_1$.

**$G_3$ from dif.com**

Same as $G_1$, except that in (b), $|k - \ell| < x_1 + 1$ is replaced by $|k - \ell| < y_1 - x_1 + 1$.

The first few P-positions for $G_2$ and $G_3$ are listed in Tables 2 and 3 respectively. In Section 4 we also prove,

**Theorem 2.** For $G_2$ and $G_3$, $P = \bigcup_{i=0}^{\infty} (A_i, B_i)$, where, for all $n \in \mathbb{Z}_{\geq 0}$, $A_n$ is given by (3), and $B_n = 2A_n$ for $G_2$; and $B_0 = 0$, and for $n \in \mathbb{Z}_{\geq 1}$, $B_n = A_n + 2^n - 1$ for $G_3$. 

Each of our games is associated with a pair of complementary sequences \( A_n, B_n \). A special case is the well-known (classical) Wythoff (1907) game. See also Berlekamp, Conway, and Guy (1982), Blass and Fraenkel (1990), Connell (1959), Coxeter (1953), Dress (1999), Fraenkel (1982), Fraenkel (1984), Fraenkel and Borosh (1973), Fraenkel and Ozery (1998), Landman (2002), Silber (1976), Silber (1977), and Yaglom and Yaglom (1967). In fact, the classical Wythoff game is the case \( f(x_1, y_1, x_0) = 1 \), whereas the generalization considered in Fraenkel (1982) is the case \( f(x_1, y_1, x_0) = t \) for any fixed \( t \in \mathbb{Z}_{>0} \). Whereas the winning strategy of Wythoff’s game is associated with sequences related to algebraic integers of the form \( \left(2 - t + \sqrt{t^2 + 4}\right)/2 \) (\( t=1 \) is the golden section), our games give rise to an infinity of sequences, some well-known, but mostly new ones.

In Section 2 we shall see that the pair of sequences of \( P \)-positions associated with \( G_1 \) is related to a “self-generating” sequence (Sloane, 1999) of Hofstadter. In Section 3 we indicate how the \( P \)-positions of \( G_2 \) are related to another well-known sequence. The central result appears in Section 4, where a general theorem is formulated and proved, that yields winning strategies for a large class of 2-pile subtraction games. Roughly speaking it states that for every 2-pile subtraction game, if its constraint function \( f \) is “positive”, “monotone” and “semi-additive”, then it has \( P \)-positions \( A_n, B_n \), where \( A_n \) satisfies (3), and \( B_n \) has an explicit form depending on \( f \). In a complementary proposition we show that positivity, monotonicity and semi-additivity are also necessary, in the sense that if any one of them is dropped, then there are constraint functions and their associated games \( G \), such that the positions claimed to be \( P \)-positions by the central result, are not \( P \)-positions for these \( G \). Theorems 1 and 2 are then deduced as a simple corollary of the central result. In Section 5 we give a random assortment of sample games with their \( P \)-positions that can be produced from the central theorem. Questions of complexity and related issues are discussed in Section 6. The epilogue in Section 7 wraps up with some concluding remarks and indications for further study.

2. The Gödel, Escher, Bach Connection

On p. 73 of Hofstadter’s (1979) famous book the reader is asked to characterize the following sequence:

\[
B'_{n \geq 0} = \{1, 3, 7, 12, 18, 26, 35, 45, 56, \ldots \}.
\]

Answer: the sequence \{2, 4, 5, 6, 8, 9, 10, 11, \ldots \} constitutes the set of differences of consecutive terms of \( B'_n \), as well as the complement with respect to \( \mathbb{Z}_{>0} \) of \( B'_n \). For our purposes it is convenient to preface 0 to the latter sequence, so we define

\[
A'_{n \geq 0} = \{0, 2, 4, 5, 6, 8, 9, 10, 11, \ldots \},
\]
which is the complement with respect to $\mathbb{Z}_{\geq 0}$ of $B'_{n}$. Now $A'_{10} = \text{mex}\{A'_{i}, B'_{i} : 0 \leq i < 10\} = 13$, so $B'_{10} = 56 + 13 = 69$. We see that in general, for all $n \in \mathbb{Z}_{\geq 0}$,

$$A'_{n} = \text{mex}\{A'_{i}, B'_{i} : 0 \leq i < n\}$$

(6)

which has the form (3), and

$$B'_{n} = B'_{n-1} + A'_{n},$$

(7)

which is similar to (4). Moreover, the following proposition shows that there is a very close relationship between the $P$-positions of the game $G_{1}$ and Hofstadter’s sequence $B'_{n}$, namely, $B'_{n}$ exceeds $B_{n}$ by 1. This can be observed by comparing the bottom row of Table 1 with $B'_{n}$.

Proposition 1. $A'_{n} = A_{n} + 1 \; (n \geq 1), \quad B'_{n} = B_{n} + 1 \; (n \geq 0)$, where $A'_{n}, B'_{n}$ are given by (6), (7) respectively, and $A_{n}, B_{n}$ by (3), (4) respectively.

Proof. We see that the assertions are true for small $n$. Suppose they hold for all $i \leq n$. Then

$$A'_{n+1} = \text{mex}\{A'_{i}, B'_{i} : 0 \leq i \leq n\} = \text{mex}\{0, A_{i} + 1, B_{i} + 1 : 0 \leq i \leq n\}. $$

Put $S'_{n} = \{0, A_{i} + 1, B_{i} + 1 : 0 \leq i \leq n\}$, $S_{n} = \{A_{i}, B_{i} : 0 \leq i \leq n\}$. If, say, the integer interval $[0, k]$ is in $S_{n}$ for some $k \in \mathbb{Z}_{\geq 0}$ and $k + 1 \notin S_{n}$, then $k + 1 \in S'_{n}$ and $k + 2 \notin S'_{n}$. It follows that $\text{mex} \; S'_{n} = A_{n+1} + 1$. Also, $B'_{n+1} = B'_{n} + A'_{n+1} = B_{n} + 1 + A_{n+1} + 1 = B_{n+1} + 1$. ■

Thus the $P$-positions of $G_{1}$ constitute a “translation by 1” of the Hofstadter sequence, that is, $B_{n+1} - B_{n} = A_{n+1} + 1$. So $A_{n} + 1$ is the difference (dif) and $A_{n}$ the complement (com) of $B_{n}$: they are products of dif.com.

3. Prouhet-Thue-Morse

It is not hard to see that the sequence $A_{n} \; (n \geq 1)$ of $G_{2}$ contains precisely all positive integers whose binary representation ends in even number of zeros. (Because of this, $G_{2}$ originates from even.com: “www.even.com is the best place to find information and sources for even”, it says on its webpage.) The sequence $A_{n}$ is also lexicographically minimal with respect to the property that the parity of number of 1’s in the binary expansion alternates. Furthermore, it is lexicographically minimal with respect to the property that the sequence is the double of its complement. If $m$ appears in $A_{n}$, then $2m$ appears in $B_{n}$. In particular, $B_{n}$ contains precisely all positive integers whose binary representation ends in an odd number of zeros (Carlitz, Scoville, and Hoggatt, 1972). The sequence

$$C_{n} = 0^{A_{1}-A_{0}} 1^{A_{2}-A_{1}} 0^{A_{3}-A_{2}} \ldots 0^{A_{2n+1}-A_{2n}} 1^{A_{2n+2}-A_{2n+1}} \ldots$$

$$= 011010011001011010010\ldots.$$
is the Prouhet-Thue-Morse sequence, which arises in many different areas of mathematics. See the charming paper (Allouche and Shallit, 1999), which also contains $A_n$, for many further properties of these sequences.

4. A Master Theorem

The three previously described games $G_1$, $G_2$, $G_3$, are special cases of an infinite family of games that we now formulate. We shall then provide a general winning strategy for this family of games and prove its validity.

**General 2-pile subtraction games**

Given two piles of tokens $(x, y)$ of sizes $x, y$, with $0 \leq x \leq y < \infty$, whose $P$-positions are $\mathcal{P} = \cup_{i=0}^{\infty}(A_i, B_i)$. Two players alternate removing tokens from the piles:

(aa) Remove any positive number of tokens from a single pile, possibly the entire pile.

(bb) Remove a positive number of tokens from each pile, say $k, \ell$, so that $|k - \ell|$ is not too large with respect to the position $(x_1, y_1)$ moved to from $(x_0, y_0)$, namely, $|k - \ell| < f(x_1, y_1, x_0)$, equivalently:

\[ |(y_0 - y_1) - (x_0 - x_1)| = |(y_0 - x_0) - (y_1 - x_1)| < f(x_1, y_1, x_0), \quad (8) \]

where the constraint function $f(x_1, y_1, x_0)$ is integer-valued and satisfies:

- **Positivity:**
  \[ f(x_1, y_1, x_0) > 0 \quad \forall y_1 \geq x_1 \geq 0 \quad \forall x_0 > x_1. \]

- **Monotonicity:**
  \[ x'_0 < x_0 \implies f(x_1, y_1, x'_0) \leq f(x_1, y_1, x_0). \]

- **Semi-additivity** (or generalized triangle inequality) on the $P$-positions, namely: for $n > m \geq 0$,
  \[ \sum_{i=0}^{m} f(A_{n-1-i}, B_{n-1-i}, A_{n-i}) \geq f(A_{n-m-1}, B_{n-m-1}, A_n). \]

The player making the move after which both piles are empty wins; the opponent loses.

In view of (8), positivity is a natural condition. Without positivity, a move of type (bb) is not even possible. Monotonicity appears to be a minimal requirement to enforce positivity. Semi-additivity is a convenient condition to have, and many functions are semi-additive. Note that $G_1$, $G_2$, $G_3$ clearly satisfy
positivity and monotonicity; $G_1$ and $G_3$, in whose functions $f$ there is no $A_n$, are clearly semi-additive; and $G_2$ is semi-additive with equality. (See also the proof of Theorems 1 and 2 at the end of this section.)

**Theorem 3.** Let $S = \bigcup_{i=0}^{\infty} (A_i, B_i)$, where, for all $n \in \mathbb{Z}_{\geq 0}$, $A_n$ is given by (3), $B_0 = 0$, and for all $n \in \mathbb{Z}_{>0}$,

$$B_n = f(A_{n-1}, B_{n-1}, A_n) + B_{n-1} - A_{n-1} + A_n.$$  

If $f$ is positive, monotone and semi-additive, then $S$ is the set of $P$-positions of a general 2-pile subtraction game with constraint function $f$.

**Proof.** The definition of $A_n$ implies directly,

$$A_n > A_{n-1}$$  

for all $n \in \mathbb{Z}_{>0}$. From (9) we have, for all $n \in \mathbb{Z}_{>0}$,

$$B_n - B_{n-1} = f(A_{n-1}, B_{n-1}, A_n) + A_n - A_{n-1},$$  

$$B_n - A_n = f(A_{n-1}, B_{n-1}, A_n) + B_{n-1} - A_{n-1}.$$  

Now $f(A_0, B_0, A_1) > 0$ by positivity, so $B_1 - B_0 \geq 2$ by (10), (11). Hence we get, by induction on $n$,

$$B_n - B_m \geq 2 \text{ for all } n > m \geq 0.$$  

Similarly we get from (12),

$$B_n - A_n > B_m - A_m \geq 0 \text{ for all } n > m \geq 0.$$  

Let $A = \bigcup_{n=1}^{\infty} A_n$ and $B = \bigcup_{n=1}^{\infty} B_n$. Then $A$ and $B$ are complementary sets of integers, i.e., $A \cup B = \mathbb{Z}_{\geq 1}$ (by (3)), and $A \cap B = \emptyset$. Indeed, if $A_n = B_m$, then $n > m$ implies that $A_n$ is the mex of a set containing $B_m = A_n$, a contradiction to the mex definition; and $1 \leq n \leq m$ is impossible since

$$B_m = f(A_{m-1}, B_{m-1}, A_m) + B_{m-1} - A_{m-1} + A_m$$  

$$\geq f(A_{m-1}, B_{m-1}, A_n) + B_{n-1} - A_{n-1} + A_n$$  

(by (10), (14) and monotonicity)  

$$> A_n \text{ (by positivity)}.$$  

Since $B_n - B_{n-1} \geq 2 \text{ for all } n \geq 1$ by (13), and since $A$ and $B$ are complementary,

$$A_n - A_{n-1} \in \{1, 2\}$$  

(15)
for all \( n \in \mathbb{Z}_{>0} \). Denote by \( \mathcal{P}' \) the set of all positions \((A_n, B_n)\) satisfying (3) and (9), and let \( \mathcal{N}' = \mathbb{Z}_{>0} \setminus \mathcal{P}' \). For showing that \( \mathcal{P}' = \mathcal{P} \) and \( \mathcal{N}' = \mathcal{N} \), it evidently suffices to show two things:

I. Every move from any \((A_n, B_n) \in \mathcal{P}'\) results in a position in the complement \(\mathcal{N}'\).

II. From every position \((x, y)\) in the complement \(\mathcal{N}'\), there is a move to some \((A_n, B_n) \in \mathcal{P}'\).

(It is useful to note that these two conditions are also necessary: (1) implies that all positions reachable in one move from a \(P\)-position are \(N\)-positions; whereas (2) shows that at least one \(P\)-position is reachable in one move from an \(N\)-position.)

I. A move of type (aa) from \((A_n, B_n) \in \mathcal{P}'\) has the form \((x, B_n)\) or \((A_n, y)\) \((x < A_n, y < B_n)\). Both are in \(\mathcal{N}'\) since the sequences \(A_n, B_n\) are strictly increasing. Suppose there is a move of type (bb): \((A_n, B_n) \rightarrow (A_j, B_j) \in \mathcal{P}'\). Then \(j < n\). Note that

\[
\begin{align*}
|(B_n - B_j) - (A_n - A_j)| &= |(B_n - A_n) - (B_j - A_j)| = (B_n - A_n) - (B_j - A_j) \\
&= (B_n - A_n) - (B_j - A_j)
\end{align*}
\]

by (14). By iterating (9) we have,

\[
\begin{align*}
(B_n - A_n) - (B_j - A_j)
&= f(A_{n-1}, B_{n-1}, A_n) + (B_{n-1} - A_{n-1}) - (B_j - A_j) \\
&= f(A_{n-1}, B_{n-1}, A_n) + f(A_{n-2}, B_{n-2}, A_{n-1}) \\
&\quad + (B_{n-2} - A_{n-2}) - (B_j - A_j) \\
&\vdots \\
&= \sum_{i=0}^{n-j-1} f(A_{n-i-1}, B_{n-i-1}, A_{n-i}) \geq f(A_j, B_j, A_n),
\end{align*}
\]

where the inequality follows from semi-additivity. Thus

\[
|(B_n - B_j) - (A_n - A_j)| \geq f(A_j, B_j, A_n),
\]

contradicting condition (bb).

II. Let \((x, y) \in \mathcal{N}' \ (0 \leq x \leq y)\). Since \(A\) and \(B\) are complementary, every \(n \in \mathbb{Z}_{>0}\) appears exactly once in exactly one of \(A\) and \(B\). Therefore we have either \(x = B_n\) or else \(x = A_n\) for some \(n \geq 0\).

(i) \(x = B_n\). Then move \(y \rightarrow A_n\). This is always possible since if \(n = 0\), then \(y > A_0 = B_0\); whereas \(A_n < B_n\) for \(n \geq 1\) by (14).
New Games Related to Old and New Sequences 375

(ii) $x = A_n$. If $y > B_n$, move $y \rightarrow B_n$. So suppose that $A_n \leq y < B_n$. Then $n \geq 1$. For any $m \in \{0, \ldots, n-1\}$ we have by (9) and by monotonicity,

$$
(B_{m+1} - A_{m+1}) - (B_m - A_m) = f(A_m, B_m, A_{m+1}) \leq f(A_m, B_m, A_n).
$$

Thus $B_m - A_m + f(A_m, B_m, A_n) \geq B_{m+1} - A_{m+1}$. Therefore the intervals $[B_m - A_m, B_m - A_m + f(A_m, B_m, A_n))$ (closed on the left, open on the right) cover $\mathbb{Z}_{\geq 0}$ for $m \geq 0$. Hence

$$
y - A_n \in [B_m - A_m, B_m - A_m + f(A_m, B_m, A_n)) \quad (16)
$$

for a smallest $m \in \{0, \ldots, n-1\}$. We then move $(x, y) \rightarrow (A_m, B_m)$. This move is legal, since:

- $m < n$. Indeed, $y - A_n < B_n - A_n = f(A_{n-1}, B_{n-1}, A_n) + B_{n-1} - A_{n-1}$. Thus $m \leq n - 1$ by (16).
- $y > B_m$. By (16), $y - A_n \geq B_m - A_m$. Hence $y - B_m \geq A_n - A_m > 0$.
- The move satisfies (bb):

$$
|y - B_m) - (x - A_m)| = |(y - A_n) - (B_m - A_m)| \\
= (y - A_n) - (B_m - A_m)
$$

where the last equality follows from (16) and our choice of $m$. We thus have $|(y - A_n) - (B_m - A_m)| = (y - A_n) - (B_m - A_m) < f(A_m, B_m, A_n)$ by (16). □

In a sense, Theorem 3 is best possible. This is enunciated below.

**Proposition 2.** There exist 2-pile subtraction games with constraint functions $f$ which lack precisely one of positivity, monotonicity or semi-additivity, such that $S \neq \mathcal{P}$, where $S = \bigcup_{i=0}^{\infty} (A_i, B_i)$, and $A_i$ satisfies (3) ($i \in \mathbb{Z}_{\geq 0}$); $B_0 = 0$, $B_i$ satisfies (9) ($i \in \mathbb{Z}_{>0}$).

**Proof.** Consider the function $f(x_1, y_1, x_0) = (x_0 - x_1)^2$. It is clearly positive and monotone. However, $(A_n - A_{n-1})^2 + (A_{n-1} - A_{n-2})^2 < (A_n - A_{n-2})^2$, no matter whether $A_n - A_{n-1} = A_{n-1} - A_{n-2} = 1$ or otherwise, so $f$ is not semi-additive. From (9) we get, $B_n = B_{n+1} + (A_n - A_{n-1})(A_n - A_{n-1} + 1)$, where $A_n$ satisfies (3). The first few values of $(A_n, B_n)$ are depicted in Table 4. Note that these are not $P$-positions: we can move $(A_n, B_n) \rightarrow (A_i, B_i)$ in many ways; e.g., $(4, 10) \rightarrow (0, 0)$ satisfies (bb).

The function $f(x_1, y_1, x_0) = [(x_1 + 1)/x_0] + 1$ is positive. Since

$$
\left(\left\lfloor \frac{A_{n-1} + 1}{A_n} \right\rfloor + 1\right) + \left(\left\lfloor \frac{A_{n-2} + 1}{A_{n-1}} \right\rfloor + 1\right) > \left\lfloor \frac{A_{n-2} + 1}{A_n} \right\rfloor + 1 = 1,
$$
Table 4. The first few values of $S$ for $f = (x_0 - x_1)^2$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>22</td>
<td>28</td>
<td>34</td>
<td>40</td>
<td>46</td>
<td>48</td>
<td>50</td>
<td>52</td>
<td>54</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 5. The first few values of $S$ for $f = \lfloor (x_1 + 1)/x_0 \rfloor + 1$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>20</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
<td>30</td>
<td>33</td>
<td>36</td>
<td>39</td>
<td>42</td>
<td>45</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 6. The first few values of $P$-positions for $f = \left(1 + (-1)^{y_1+1}\right)x_1/2$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_n$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>$B_n$</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>12</td>
<td>13</td>
<td>21</td>
<td>30</td>
<td>31</td>
<td>42</td>
<td>45</td>
<td>60</td>
<td>61</td>
<td>78</td>
<td>79</td>
<td>98</td>
<td>99</td>
</tr>
</tbody>
</table>

it is also semi-additive. But it is not monotone. From (9), $B_n = B_{n-1} - A_{n-1} + A_n + \lfloor (A_{n-1} + 1)/A_n \rfloor + 1$. The first few values of $S = \bigcup_{n=0}^{\infty}(A_n, B_n)$ are shown in Table 5. The game-position $(4, 7) \notin S$, but it cannot be moved into $S$. Hence $S \neq P$. (Incidentally, note that the sequence $B_n$ consists of all nonnegative multiples of 3.)

Lastly, consider $f(x_1, y_1, x_0) = \left(1 + (-1)^{y_1+1}\right)x_1/2$. We see easily that $f$ is semi-additive, and it is trivially monotone. But whenever $y_1$ is even, $f$ is not positive. We have, $B_n = A_n + B_{n-1} - \left(1 + (-1)^{B_{n-1}}\right)A_{n-1}/2$. Table 6 shows the first few $S$-positions. These are not $P$-positions: The position $(10, 29) \notin S$, cannot be moved into any position in $S$. ■

**Proof of Theorems 1 and 2.** The function $f(x_1, y_1, x_0) = x_1 + 1$ is clearly positive. Monotonicity is satisfied trivially. It is also clear that $f$ is semi-additive. The function $f(x_1, y_1, x_0) = x_0 - x_1$ is positive, since $x_0 > x_1$. It’s also monotone. Since $(A_{n+1} - A_n) + (A_n - A_{n-1}) = A_{n+1} - A_{n-1}$, we see that $f$ is semi-additive. Finally, the function $f(x_1, y_1, x_0) = y_1 - x_1 + 1$
New Games Related to Old and New Sequences

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_n)</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>(B_n)</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>12</td>
<td>18</td>
<td>25</td>
<td>35</td>
<td>45</td>
<td>56</td>
<td>68</td>
<td>83</td>
<td>98</td>
<td>114</td>
<td>131</td>
<td>149</td>
<td>170</td>
<td>191</td>
</tr>
</tbody>
</table>

Table 7. The first few values of \(S\) for \(f = x_1 - [(x_1 + 1)/x_0] + 2\).

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_n)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>(B_n)</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>18</td>
<td>22</td>
<td>26</td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>46</td>
<td>50</td>
<td>54</td>
<td>58</td>
<td>62</td>
<td>68</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 8. The first few values of \(S\) for \(f = x_0 - x_1 + 2\).

is positive for all \(x_1 \leq y_1\) and is trivially monotone. It is also semi-additive. Thus by Theorem 3 we have for \(G_1\), \(B_n = A_{n-1} + 1 + B_{n-1} - A_{n-1} + A_n = B_{n-1} + A_n + 1\), as stated in Theorem 1. For \(G_2\), (9) implies, \(B_n = A_n - A_{n-1} + B_{n-1} - A_{n-1} + A_n = 2A_n - 2A_{n-1} + B_{n-1} = 2A_n\), where the last equality follows by induction on \(n\). For \(G_3\), \(B_n = B_{n-1} - A_{n-1} + 1 + B_{n-1} - A_{n-1} + A_n = 2(B_{n-1} - A_{n-1}) + A_n + 1 = A_n + 2^n - 1\). Again the last equality follows by induction. ■

5. Further Sample Games

For the examples below, we leave it to the reader to verify positivity, monotonicity and semi-additivity of \(f\). Some of these examples are elaborated on in the next two sections.

**Example 1.** \(f(x_1, y_1, x_0) = x_1 - [(x_1 + 1)/x_0] + 2\). Then \(B_n = B_{n-1} + A_n - [(A_{n-1} + 1)/A_n] + 2\). The first few \(P\)-positions are depicted in Table 7.

**Example 2.** \(f(x_1, y_1, x_0) = x_0 - x_1 + 2\). Then \(B_n = B_{n-1} + 2(A_n - A_{n-1} + 1)\). See Table 8 for the first few \(P\)-positions.

**Example 3.** \(f(x_1, y_1, x_0) = (-1)^{y_1} - (-1)^{x_1} + 3\). Then \(B_n = B_{n-1} - A_{n-1} + A_n + (-1)^{B_{n-1}} - (-1)^{A_{n-1}} + 3\). See Table 9 for the first few \(P\)-positions.

**Example 4.** \(f(x_1, y_1, x_0) = x_1 (1 + (-1)^{x_1}) + 1\). This leads to \(B_n = B_{n-1} + (-1)^{A_{n-1}} A_{n-1} + A_n + 1\). Table 10 exhibits the first few \(P\)-positions.
A.S. Fraenkel

6. Computational Complexity Issues

What is the computational complexity of computing the winning strategy for our games? Given a position \((x, y)\) with \(0 \leq x \leq y < \infty\), the statement of Theorem 3 enables us to compute the table of \(P\)-positions. It suffices to compute it up to the smallest \(n = n_0\) such that \(A_{n_0} \geq x\), and thus determine whether \((x, y) \in \mathcal{P}\) or in \(\mathcal{N}\). The proof of Theorem 3 then enables us, if \((x, y) \in \mathcal{N}\), to make a winning move to a position in \(\mathcal{P}\). The latter part of the strategy, that of making a winning move, is clearly polynomial. The first part, determining whether or not \((x, y) \in \mathcal{P}\) is linear in \(x\), since \(A_{n_0} \leq 2x\) by (15).

Our games, however, are succinct, i.e., the input size is \(\Omega(\log x)\) rather than \(\Omega(x)\) (assuming that \(y\) is bounded by a polynomial in \(x\)). Thus their complexity is not obvious a priori. Even if the \(B_n\)-sequence grows exponentially, polynomiality of the strategy does not necessarily follow. For example, I do not know whether the sequence \(B_n\) of \(G_3\) can be computed polynomially.

Special sequences are known to be computable polynomially. For example, consider the numeration system with bases defined by the recurrence \(u_n = (s + t - 1)u_{n-1} + su_{n-2}\) \((n \geq 1)\), where \(s, t \in \mathbb{Z}_{>0}\), with initial conditions \(u_{-1} = 1/s\), \(u_0 = 1\). It follows from Fraenkel (1985) that every positive integer \(N\) has a unique representation of the form \(N = \sum_{i \geq 0} d_i u_i\), with digits \(d_i \in \{0, \ldots, s + t - 1\}\), such that \(d_{i+1} = s + t - 1 \implies d_i < s\) for all \(i \in \mathbb{Z}_{\geq 0}\). The representation of the first few entries for the special case \(s = 2\), \(t = 2\), is depicted in Table 10.

If we compare Table 11 with Table 8, we might note the following two properties:

| \(n\) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|    |
| \(A_n\) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 11 | 12 | 13 | 15 | 16 | 17 | 18 | 19 |
| \(B_n\) | 0 | 4 | 10 | 14 | 21 | 25 | 27 | 31 | 33 | 38 | 44 | 48 | 55 | 59 | 61 | 65 | 67 |    |

Table 9. The first few values of \(S\) for \(f = (-1)^y - (-1)^{x_1} + 3\).

| \(n\) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|    |
| \(A_n\) | 0 | 1 | 3 | 4 | 6 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 19 | 20 |
| \(B_n\) | 0 | 2 | 5 | 7 | 18 | 33 | 42 | 44 | 66 | 68 | 94 | 96 | 136 | 138 | 172 | 175 | 177 |

Table 10. The first few values of \(S\) for \(f = x_1 (1 + (-1)^{x_1}) + 1\).
New Games Related to Old and New Sequences  

<table>
<thead>
<tr>
<th>50</th>
<th>14</th>
<th>4</th>
<th>1</th>
<th>n</th>
<th>14</th>
<th>4</th>
<th>1</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>31</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>32</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>33</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>34</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>35</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>36</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>37</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>38</td>
<td>2</td>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>39</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>40</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>41</td>
<td>2</td>
<td>3</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>42</td>
<td>3</td>
<td>0</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>43</td>
<td>3</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>44</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>45</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>46</td>
<td>1</td>
<td>0</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>47</td>
<td>1</td>
<td>0</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>48</td>
<td>1</td>
<td>1</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>49</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>1</td>
<td>1</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>51</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>52</td>
<td>1</td>
<td>2</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>53</td>
<td>1</td>
<td>2</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>54</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>55</td>
<td>1</td>
<td>2</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>56</td>
<td>1</td>
<td>3</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>57</td>
<td>1</td>
<td>3</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>58</td>
<td>2</td>
<td>0</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>59</td>
<td>2</td>
<td>0</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>60</td>
<td>2</td>
<td>0</td>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11. Representation of the first few integers in a special numeration system.

- All the $A_n$ have representations ending in an even number of Os, and all the $B_n$ have representations ending in an odd number of Os.
- For every $(A_n, B_n) \in \mathcal{P}$, the representation of $B_n$ is the "left shift" of the representation of $A_n$.

Thus $(1, 4)$ of Table 11 has representation $(1, 10)$, and $(6, 22)$ has representation $(12, 120)$: 10 is the "left shift" of 1, 120 the left shift of 12.

These properties hold, in fact, in general for Example 2, which is a special case of another family of sequences and games analyzed in Fraenkel (1998). They enable one to win in polynomial time for that family.

However, we do not even know whether there are NP-hard sequences. A case in point is the infinite family of octal games (Guy and Smith, 1956; Berlekamp,
Conway, and Guy, 1982, ch. 4), even for the subfamily where there are only
finitely many nonzero octal digits. Some octal games have been shown to
have polynomial strategies (see, e.g., Gangolli and Plambeck, 1989) but the
complexity of most is unknown.

We mention very briefly other relevant complexities. They include Kol-
mogorov complexity, subword complexity, palindrome complexity, and, we
might add, squares complexity. The subword complexity $c(n)$ of a sequence $S$
is the number of distinct words of length $n$ occurring in $S$. In Allouche et al.
(2003), this notion is attributed to Ehrenfeucht, Lee, and Rozenberg (1975).
Surveys can be found in Allouche (1994), Ferenczi (1999), Ferenczi and Kása
(1999). The palindrome complexity $p(n)$ of $S$ is the number of distinct palin-
ンドromes of length $n$ in $S$. See, e.g., Damanik and Zare (2000), and Allouche
and Shallit (2003). Define the squares complexity $s(n)$ of $S$ as the number
of distinct squares of length $n$ in $S$. Thus the result of Fraenkel and Simpson
(1995) implies that there are binary sequences for which $s(2) = 2$, $s(4) = 1$,$s(2k) = 0$ for all $k > 2$. There is also the notion of program complexity (Daley,
1973, 1974, 1975) concerning the complexity of computing a sequence, which
is related to Kolmogorov (1968) complexity.

7. Epilogue

We have defined an infinite class of 2-pile subtraction games with two types
of moves: (aaa) remove any positive number from a single pile; (bbb) remove
$k > 0$ from one pile, $\ell > 0$ from the other. This move is restricted by the
requirement $|k - \ell| < f$, where $f$ is a positive real-valued function. We
have shown that a pair $A_n, B_n$ of judiciously chosen complementary sequences
constitutes the set of $P$-positions if and only if $f$ is monotone and semi-additive.

As we have pointed out, the generalized Wythoff game (Fraenkel, 1982) is
a special case of the family of games considered here. It has the property that
a polynomial strategy can be given by using a natural numeration system, and
noting that the $A_n$ members are characterized by ending in an even number of
zeros in that representation, and the $B_n$ being their left shifts. A similar situation
exists for $G_2$, but with the standard binary representation as numeration system.
With the game in Example 2, an essentially different numeration system (see
Fraenkel, 1998) can be associated to the same effect.

Further studies

1. With which games can we associate an appropriate numeration system so
as to establish a polynomial strategy?
2. Extend the games in a natural way to handle more than two piles. For
Wythoff’s game, I have a conjecture (see Guy and Nowakowski, 2002, Prob-
lem 53; Fraenkel, 2003, Section 5).
3. Compute the Sprague-Grundy function $g$ for the games, which will enable to play sums of games. For Wythoff’s game this is an as yet unsolved problem, though eventual additive periodicity has been proved (Dress, Flammenkamp, and Pink, 1999; Landman, 2002).

4. Compute a strategy for the games when played in misère version, i.e., the player making the last move loses. This is easy for Wythoff’s game (see Berlekamp, Conway, and Guy, 1982, ch. 13).

5. We have already mentioned the question of the polynomiality of the strategy. Is there a 2-pile subtraction game that is Pspace-complete?

6. Computation of complexities of $P$-positions sequences, such as Kolmogorov-, program-, subword-, palindrome-, squares-complexities. For the $A_n$-sequence of Example 2, the subword complexity was computed in Fraenkel, Seeman, and Simpson (2001).

7. Make an about-face: begin with pairs of known complementary sequences, and design matching 2-pile subtraction games.

References

http://www.combinatorics.org/
http://www.combinatorics.org/
W.A. Wythoff, A modification of the game of Nim, Nieuw Arch. Wisk. 7 (1907) 199–202.
Author Index

ANDRIST, Rafael B. .................... 65
BILLINGS, Darse ...................... 231
BJÖRNSSON, Yngvi ............ 193, 231, 261
BOLOGNESI, Andrea .............. 325
BOUZY, Bruno ....................... 159
BRATKO, Ivan ......................... 33
BURCH, Neil .................... 193
BURO, Michael .............. 1, 19, 361
CAZENAVE, Tristan ............. 109, 343
CIANCARINI, Paolo ............. 325
DODGEN, Gilbert .................. 211
DONKERS, H. (Jeroen) H.L.M. ... 309
ENZENBERGER, Markus ............ 97
FRAENKEL, Aviezri S. ........... 367
GOMBOC, Dave ....................... 1
HAWORTH, Guy M'C ................ 65, 81
HAYWARD, Ryan ..................... 261
HELMSTETTER, Bernard ........... 159, 343
HERIK, H. Jaap van den ........ 143, 249, 309
HEINZ, Ernst A. ................... 45
JIANG, Albert X. .................... 19
JOHANSON, Michael ............... 261
KAN, Morgan ......................... 261
KANEKO, Tomoyuki ............... 279
Kawai, Satoru ....................... 279

KISHIMOTO, Akihiro ............. 125
KONONENKO, Igor ............... 33
NAKAMURA, Katsuhiko .......... 175
LAKE, Robert ....................... 193
LU, Paul ........................... 193
LIEBERUM, Jens .................... 299
MARSLAND, T. Anthony .......... 1
MÜLLER, Martin ..................... 125
PO, Nathan ......................... 261
RIJSWIJCK, Jack van ............. 261
SADIKOV, Aleksander ............ 33
SCHAEFFER, Jonathan ............ 193
SUTPHEN, Steve .................... 193
TAMPLIN, John A. .................. 81
TRICE, Edward ...................... 211
UITERWIJK, Jos W.H.M. ........ 143, 249, 309
VILÀ, Ricard ....................... 109
WERF, Erik C.D. van der .......... 143
WINANDS, Mark H.M. ............ 249
YAMAGUCHI, Kazunori .......... 279