H. Jaap Van den Herik
Pieter Spronck (Eds.)

LNCS 6048

Advances in Computer Games

12th International Conference, ACG 2009
Pamplona, Spain, May 2009
Revised Papers

Springer

H. Jaap Van den Herik
Pieter Spronck (Eds.)

Advances in Computer Games

12th International Conference, ACG 2009
Pamplona, Spain, May 11-13, 2009
Revised Papers

Springer

Volume Editors

H. Jaap Van den Herik
Pieter Spronck
Tilburg centre for Creative Computing (TiCC)
Tilburg University
P.O. Box 90153
5000 LE Tilburg, The Netherlands
E-mail: {h.j.vdnherik, p.spronck}@uvt.nl

Library of Congress Control Number: 2010926225

CR Subject Classification (1998): F.2, F.1, I.2, G.2, I.4, C.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-12992-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-12992-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This book contains the papers of the 12th Advances in Computer Games Conference (ACG 2009) held in Pamplona, Spain. The conference took place during May 11–13, 2009 in conjunction with the 13^{th} Computer Olympiad and the 16^{th} World Computer Chess Championship.

The Advances in Computer Games conference series is a major international forum for researchers and developers interested in all aspects of artificial intelligence and computer game playing. The Pamplona conference was definitively characterized by fresh ideas for a large variety of games.

The Program Committee (PC) received 41 submissions. Each paper was initially sent to at least three referees. If conflicting views on a paper were reported, it was sent to an additional referee. Out of the 41 submissions, one was withdrawn before the final decisions were made. With the help of many referees (see after the preface), the PC accepted 20 papers for presentation at the conference and publication in these proceedings.

The above-mentioned set of 20 papers covers a wide range of computer games. The papers deal with many different research topics. We mention: Monte-Carlo Tree Search, Bayesian Modeling, Selective Search, the Use of Brute Force, Conflict Resolution, Solving Games, Optimization, Concept Discovery, Incongruity Theory, and Data Assurance.

The 17 games that are discussed are: Arimaa, Breakthrough, Chess, Chinese Chess, Go, Havannah, Hex, Kakuro, k-in-a-Row, Kriegspiel, LOA, 3 x n AB Games, Poker, Roshambo, Settlers of Catan, Sum of Switches, and Video Games.

We hope that the readers will enjoy the research efforts performed by the authors. Below we provide a brief characterization of the 20 contributions, in the order in which they are published in the book.

"Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search," by Guillaume Chaslot, Christophe Fiter, Jeap-Baptiste Hoock, Arpad Rimmel, and Oliver Teytaud, presents a new exploration term, which is important in the trade-off between exploitation and exploration. Although the new term improves the Monte-Carlo Tree Search considerably, experiments show that some important situations (semeais, nakade) are still not solved. Therefore, the authors offer three other important improvements. The contributions is a joy to read and provides ample insights into the underlying ideas of the Go program MoGo.

"A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm" is authored by Markus Enzenberger and Martin Müller. The contribution focuses on efficient parallelization. The ideas on a lock-free multithreaded Monte-Carlo Tree Search aim at taking advantages of the memory model of the AI-32 and Intel-64 CPU architectures. The algorithm is applied in the Fuego Go program and has improved the scalability considerably.

"Monte-Carlo Tree Search in Settlers of Catan," by Ostván Szita, Guillaume Chaslot, and Pieter Spronck, describes a successful application of MCTS in multi-player strategic decision making. The authors use the non-deterministic board game Settlers of Catan for their experiments. They show that providing a game-playing algorithm with (limited) domain knowledge can improve the playing strength substantially. Two techniques that are discussed and tested are: (1) using non-uniform sampling in the Monte-Carlo simulation phase and (2) modifying the statistics stored in the game tree.

"Evaluation Function-Based Monte-Carlo LOA" is written by Mark H.M. Winands and Yngvi Björnsson. The paper investigates how to use a positional evaluation function in a Monte-Carlo simulation-based LOA program (ML-LOA). Four different simulations strategies are designed: (1) Evaluation Cutt-Off, (2) Corrective, (3) Greedy, and (4) Mixed. Experimental results reveal that the Mixed strategy is the best among them. This strategy draws the moves randomly based on their transition probabilities in the first part of a simulation, but selects them based on their evaluation score in the second part of a simulation.

"Monte-Carlo Kakuro" by Tristan Cazenave is a one-person game that consists in filling a grid with integers that sum up to predefined values. Kakuro can be modeled as a constraint satisfaction problem. The idea is to investigate whether Monte-Carlo methods can improve the traditional search methods. Therefore, the author compares (1) Forward Checking, (2) Iterative Sampling and (3)Nested Monte-Carlo Search. The best results are produced by Nested Monte-Carlo search at level 2.

"A Study of UCT and Its Enhancements in an Artificial Game" is authored by David Tom and Martin Müller. The authors focus on a simple abstract game called the Sum of Switches (SOS). In this framework, a series of experiments with UCT and RAVE are performed. By enhancing the algorithm and fine-tuning the parameters, the algorithmic design is able to play significantly stronger without requiring more samples.

"Creating an Upper-Confidence Tree Program for Havannah, " by F. Teytaud and O. Teytaud, presents another proof of the general applicability of MCTS by testing the techniques on the Havannah game. The authors investigate Bernstein's formula, the success role of UCT, the efficiency of RAVE, and progressive widening. The outcome is quite positive in all four subdomains.

"Randomized Parallel Proof-Number Search," by Jahn Takeshi Saito, Mark H.M. Winands, and H.Jaap van den Herik, describes a new technique for parallelizing Proof Number Search (PNS) on multi-core systems with shared memory. The parallelization is based on randomizing the move selection of multiple threats, which operate on the same search tree. Experiments show that RP-PNS scales well. Four directions for future research are given.

"Hex, Braids, the Crossing Rule and XH-Search," written by Philip Henderson, Broderik Arneson, and Ryan B. Hayward, proposes XH-search, a Hex connection finding algorithm. XH-search extends Anshelevich's H-search by incorporating a new crossing rule to find braids, connections built from overlapping subconnections. XH-search is efficient and easily implemented.

"Performance and Prediction: Bayesian Modelling of Fallible Choice in Chess" is a contribution by Guy Haworth, Ken Regan, and Giuseppe Di Fatta. The authors focus on the human factor as is evidently expressed in games, such as Roshambo and Poker. They investigate (1) assessing the skill of a player, and (2) predicting the behavior of a player. For these two tasks they use Bayesian inferences. The techniques so developed enable the authors to address hot topics, such as the stability of the rating scale, the comparison of players of different eras, and controversial incidents possibly involving fraud. The last issue, for instance, discusses clandestine use of computer advice during competitions.

"Plans, Patterns and Move Categories Guiding a Highly Selective Search" written by Gerhard Trippen. New ideas for an Arimaa-playing program RAT are presented. RAT starts with a positional evaluation of the current position. A directed position graph based on pattern matching decides which plan of a given set of plans should be followed. The plan then dictates what types of moves can be chosen. Leaf nodes are evaluated only by a straightforward material evaluation. The highly selective search looks, on average, at only five moves out of 5,000 to over 40,000 possible moves in a middle game position.

"6-Man Chess and Zugzwangs" by Eiko Bleicher and Guy Haworth. They review zugzwang positions where having the move is a disadvantage. An outcome of the review is the observation that the definition of *zugzwang* should be revisited, if only because the presence of *en passent capture* moves gives rise to three, new, asymmetric types of zugzwang. With these three new types, the total number of types is now six. Moreover, there are no other types.

"Solving Kriegspiel Endings with Brute Force: The Case of KR vs K" is a contribution by Paolo Ciancarini and Gian Piero Favini. The paper proposes the solution of the KRK endgame in Kriegspiel. Using brute force and a suitable data representation, one can achieve perfect play, with perfection meaning fastest checkmate in the worst case and without making any assumptions on the opponent. The longest forced mate in KRK is 41. The KRK tablebase occupies about 80 megabytes of hard disk space. On average, the program has to examine 25,000 metapositions to find the compatible candidate with the shortest route to mate.

"Conflict Resolution of Chinese Chess Endgame Knowledge Base," written by Bon-Nian Chen, Pangfang Liu, Shun-Chin Hsu, and Tsan-sheng Hsu, proposes an autonomic strategy to construct a large set of endgame heuristics, which help to construct an endgame database. A conflict resolution strategy eliminates the conflicts among the constructed heuristic databases. The set of databases is called *endgame knowledge base.* The authors experimentally establish that the correctness of the constructed endgame knowledge base so obtained is sufficiently high for practical use.

"On Drawn k-in-a-Row Games," by Sheng-Hao Chiang, I-Chen Wu, and Ping-Hung Lin, continues the research on a generalized family of k-in-a-row games. The paper simplifies the family to *Connect (k, p).* Two players alternately place p stones on empty squares of an infinite board in each turn. The player who first obtains k connective stones of the own color horizontally, vertically, or diagonally wins the game. A *Connect*(k, p) game is drawn if both players have no winning strategy. Given p, the authors derive the value $k_{draw}(p)$, such that *Connect*$(k_{draw}(p), p)$ is drawn, as follows. (1) k_{draw} (2)=11. (2) For all $p \geq 3$, $k_{draw}(p) = 3p + 3d + 8$, where d is a logarithmic function of p. So, the ratio $k_{draw}(p)/p$ is approximate to 3 for sufficiently large p. To their knowledge, the $k_{draw}(p)$ are currently the smallest for all $2 \leq p < 1000$, except for p=3.

"Optimal Analyses for 3 × n AB Games in the Worst Case," is written by Li-Te Huang and Shun-Shii Lin. The paper observes that by the complex behavior of deductive genes, tree-search approaches are often adopted to find optimal strategies. In the paper, a generalized version of deductive games, called $3 \times n$ AB games, is introduced. Here, traditional tree-search approaches are not appropriate for solving this problem. Therefore a new method is developed called *structural reduction*. A worthwhile formula for calculating the optimal numbers of guesses required for arbitrary values of n is derived and proven to be final.

Automated Discovery of Search-Extension Features is a contribution by Pálmi Skowronski, Yngvi Björnsson, and Mark H.M. Winands. The authors focus on selective search extentions. Usually, it is a manual trial-and-error task. Automating the task potentially enables the discovery of both more complex and more effective move categories. The introduction of *Gradual Focus* leads to more refined new move categories. Empirical data are presented for the game Breakthrough, showing that Gradual Focus looks at a number of combinations that is two orders of magnitude fewer than a brute-force method, while preserving adequate precision and recall.

"Deriving Concepts and Strategies from Chess Tablebases," by Matej Guid, Martin Možina, Aleksander Sadikov, and Ivan Bratko, is an actual AI challenge. A positive outcome on the human understandability of the concepts and strategies would be a milestone. The authors focus on the well-known KBNK endgame. They develop an approach that combines specialized minimax search with argument-based machine learning (ABML). In the opinion of chess coaches who commented on the derived strategy, the tutorial presentation of this strategy is appropriate for teaching chess students to play this ending.

"Incongruity-Based Adaptive Game Balancing" is a contribution by Giel van Lankveld, Pieter Spronck, Jaap van den Herik, and Matthias Rauterberg. The authors focus on the entertainment value of a game for players of different skill levels. They investigate a way of automatically adopting a game's balance. The idea of adopting the balance is based on the theory of incongruity. The theory is tested for three difficult settings. Owing to the implementation of this theory it can be avoided that a game becomes *boring* or *frustrating*.

"Data Assurance in Opaque Computations," by Joe Hurd and Guy Haworth, examines the correctness of endgame data for multiple perspectives. The issue of defining a data model for a chess endgame and the systems engineering responses to that issue are described. A structured survey has been carried out of the intrinsic challenges and complexity of creating endgame data by reviewing (1) the past pattern of errors, (2) errors crept in during work in progress, (3) errors surfacing in publications, and (4) errors occurring after the data were generated. Three learning points are given.

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 for their help. Moreover, the organizers of the three events in Pamplona (see the beginning of this preface) have contributed substantially by bringing the researchers together. Without much emphasis, we recognize the work by the committees as essential for this publication. Finally, the editors happily recognize the generous sponsors Gobierno de Navarra, Ayuntamiento de Pamplona Iruñeko Udala, Centro Europeo de Empresas e Innovación Navarra, ChessBase, Diario de Navarra, Federación Navarra de Ajedrez, Fundetec, ICGA, Navarmedia, Respuesta Digital, TiCC (Tilburg University), and Universidad Pública de Navarra.

January 2010

Jaap van den Herik
Pieter Spronck

Organization

Executive Committee

Editors	H. Jaap van den Herik Pieter Spronck
Program Co-chairs	H. Jaap van den Herik Pieter Spronck

Organizing Committee

H.Jaap van den Herik(Chair)
Pieter Spronck(Co-chair)
Aitor Gonzálex VanderSluys (Local Chair)
Carlos Urtasun Estanga (Local Co-chair)
Johanna W. Hellemons
Giel van Lankveld

List of Sponsors

Gobierno de Navarra
Ayuntamiento de Pamplona Iruñeko Udala
Centro Europeo de Empresas e Innovación Navarra
ChessBase
Diario de Navarra
Federación Navarra de Ajedrez
Fundetec
ICGA
Navarmedia
Respuesta Digital
TiCC, Tilburg University
Universidad Pública de Navarra

Program Committee

Ingo Althöfer	Rémi Coulom	Tsuyoshi Hashimoto
Yngvi Björnsson	Jeroen Donkers	Guy Haworth
Ivan Bratko	Haw-ren Fang	Ryan Hayward
Tristan Cazenave	Aviezri Fraenkel	Jaap van den Herik
Keh-Hsun Chen	James Glenn	Graham Kendall
Paolo Ciancarini	Michael Greenspan	Clyde Kruskal

Richard Lorenz	Matthias Rauterberg	Gerald Tesauro
Ulf Lorenz	Jonathan Schaeffer	Jos Uiterwijk
Martin Müller	Pieter Spronck	Mark Winands
Jacques Pitrat	Nathan Sturtevant	Jan van Zanten

Additional Referees

David Fotland	Dap Hartman	Jahn Saito
Philip Henderson	Tsan-Sheng Hsu	Yaron Shoham
Maarten Schadd	Han-Shen Huang	Erik van der Werf
Bruno Bouzy	Hiroyuki Iida	Peter Emde Boas
Guillaume Chaslot	Christian Posthoff	Gert Vriend
Omid David	Akihiro Kishimoto	Hans Weigand
Matej Guid	Aleksander Sadikov	Georgios Yannakakis

Table of Contents

Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search

Guillaume Chaslot[1], Christophe Fiter[2], Jean-Baptiste Hoock[2], Arpad Rimmel[2], and Olivier Teytaud[2]

[1] Games and AI Group, MICC, Faculty of Humanities and Sciences, Universiteit Maastricht, Maastricht, The Netherlands
[2] TAO (Inria), LRI, UMR 8623 (CNRS - Univ. Paris-Sud), bat 490 Univ. Paris-Sud 91405 Orsay, France
teytaud@lri.fr

Abstract. We present a new exploration term, more efficient than classical UCT-like exploration terms. It combines efficiently expert rules, patterns extracted from datasets, All-Moves-As-First values, and classical online values. As this improved bandit formula does not solve several important situations (semeais, nakade) in computer Go, we present three other important improvements which are central in the recent progress of our program MoGo.

– We show an expert-based improvement of Monte-Carlo simulations for *nakade* situations; we also emphasize some limitations of this modification.
– We show a technique which preserves diversity in the Monte-Carlo simulation, which greatly improves the results in 19x19.
– Whereas the UCB-based exploration term is not efficient in MoGo, we show a new exploration term which is highly efficient in MoGo.

MoGo recently won a game with handicap 7 against a 9Dan Pro player, Zhou JunXun, winner of the LG Cup 2007, and a game with handicap 6 against a 1Dan pro player, Li-Chen Chien.[1]

1 Introduction

Monte-Carlo Tree Search (MCTS [1,2,3]) is a recent tool for difficult planning tasks. Impressive results have already been produced in the case of the game of Go [2,4].

MCTS consists in building a tree, in which nodes are situations of the considered environment and the branches are actions that can be taken by an agent. The main point in MCTS is that the tree is highly unbalanced, with a strong bias in favor of important parts of the tree. The focus is on the parts of the tree in which the expected gain is the highest. For estimating which situation should be further analyzed, several algorithms have been proposed. We mention three

[1] A preliminary version of this work was presented at the EWRL workshop, without proceedings.

of them. UCT[3] (Upper Confidence bounds applied to Trees) focuses on the proportion of winning simulation plus an uncertainty measure. AMAF [5,6,4] (All Moves As First, also termed RAVE for Rapid Action-Value Estimates in the MCTS context) focuses on a compromise between UCT and heuristic information extracted from the simulations. BAST[7] (Bandit Algorithm for Search in Tree) uses UCB-like bounds modified through the overall number of nodes in the tree. Other related algorithms have been proposed as in [1], essentially using a decreasing impact of a heuristic (pattern-dependent) bias as the number of simulations increases. In all these cases, the idea is to bias random simulations with the help of statistics.

In the context of the game of Go[2], the nodes are equipped with one Go board configuration, and with statistics, typically the number of won and lost games in the simulations started from this node (the RAVE computations require some more complex statistics). MCTS uses these statistics in order to expand iteratively the tree in the regions where the expected reward is maximal. After each simulation from the current position (the root) until the end of the game, the win and loss statistics are updated in every node visited by the simulation, and a new node corresponding to the first new situation of the simulation is created. The scheme is described in Algorithm 1.

Algorithm 1. Overview of Monte-Carlo Tree Search

```
Initialize the tree $T$ to only one node, equipped with the current situation.
while There is time left do
    Simulate one game $g$ from the root of the tree to a final position, choosing moves
    as follows:
    Bandit part: for a situation in $T$, choose the move with maximal score.
    MC part: for a situation out of $T$, choose the move according to Algorithm 2.
    Update the win/loss statistics in all situations of $T$ visited by $g$.
    Add in $T$ the first situation of $g$ which is not yet in $T$.
end while
Return the move simulated most often from the root of $T$.
```

The reader is referred to [1,2,3,4,8] for a detailed presentation of MCTS techniques and various scores. We will propose our current bandit formula in section 2. (Section 3 contains our new ideas and Section 4 our conclusions).

The function used for taking decisions out of the tree (i.e., the so-called Monte-Carlo part, MC) is defined in Algorithm 2. An atari occurs when a string (a group of stones) can be captured in one move. Some Go knowledge has been added in this part in the form of 3×3 expert designed patterns in order to play more meaningful games.

However, some bottlenecks appear in MCTS. In spite of the many improvements in the bandit formula, there are still situations which are poorly handled by MCTS. For instance, MCTS uses a bandit formula for moves early in the

[2] Definitions of the different Go terms used in this article can be found on the web site http://senseis.xmp.net/

Algorithm 2. Algorithm for choosing a move in MC simulations, for the Go

```
if the last move is an atari, then
   Save the stones which are in atari if possible (this is checked by liberty count).
else
   if there is an empty location among the 8 locations around the last move which
   matches a pattern then
      Sequential move: play randomly uniformly in one of these locations.
   else
      if there is a move which captures stones then
         Capture move: capture stones.
      else
         if there is a legal move then
            Legal move: play randomly a legal move
         else
            Return pass.
         end if
      end if
   end if
end if
```

tree, but cannot establish the long-term effects which involve the behavior of the simulations far from the root. The situations which are to be clarified at the very end should therefore be included in the Monte-Carlo part and not in the Bandit part. Hence, we propose three improvements in the MC part.

1. Diversity preservation (see Section 3.1).
2. Nakade refinements (see Section 3.2).
3. Elements around the Semeai (see Section 3.3).

2 Combining Offline, Transient, Online Learnings and Expert Knowledge, with an Exploration Term

In this section we present how we combine online learning (bandit module), transient learning (RAVE values), expert knowledge (detailed below), and offline pattern-information. RAVE values are presented in [4]. We point out that this combination is far from being straightforward: due to the subtle equilibrium between online learning (i.e., naive success rates of moves), transient learning (RAVE values), and offline values, the first experiments were highly negative. It was only after careful tuning of parameters[3] that the experiments became conclusive.

The score for a decision d (i.e., a legal move) is as follows.

$$score(d) = \alpha \underbrace{\hat{p}(d)}_{Online} + \beta \underbrace{\widehat{\hat{p}}(move)}_{Transient} + (\gamma + \frac{C}{\log(2+n(d))}) \underbrace{H(d)}_{Offline} \tag{1}$$

[3] We used both manual tuning and cross-entropy methods. Parallelization was highly helpful for this.

Here the coefficients α, β, γ and C are empirically tuned coefficients depending on $n(d)$ (number of simulations of the decision d) and n (number of simulations of the current board). Below we provide the formulas for α, β, γ.

$$\beta = \frac{\#\{rave\ sims\}}{\#\{rave\ sims\} + \#\{sims\} + c_1\#\{sims\}\#\{rave\ sims\}} \tag{2}$$

$$\gamma = \frac{c_2}{\#\{rave\ sims\}} \tag{3}$$

$$\alpha = 1 - \beta - \gamma \tag{4}$$

In (2) and (3), $\#\{rave\ sims\}$ is the number of Rave-simulations, $\#\{sims\}$ is the number of simulations; C, c_1, and c_2 are empirically tuned. For the sake of completeness, we precise that C, c_1, and c_2 depend on the board size. Moreover, they are not the same in the root of the tree during the beginning of the thinking time, in the root of the tree during the end of the thinking time, and in other nodes. Also, formula (1) is computed most often by an approximated (faster) formula, and only sometimes by the complete formula - it was empirically found that the constants should not be the same in both cases. We admit that all these local engineering improvements make the formula quite unclear. Our take-home message is mainly that MoGo has good results with $\alpha + \beta + \gamma = 1$, $\gamma \simeq c_2/\#\{rave\ sims\}$ and with the logarithmic formula $C/\log(2 + n(d))$ for progressive unpruning. These rules imply that the most important part is:

- initially, the offline learning,
- later, the transient learning (RAVE values),
- and that eventually, only the "real" statistics matter.

The offline part, $H(d)$ is the sum of two terms: patterns, as in [1,9,10], and rules, which are detailed below:

- capture moves (in particular, a string contiguous to a new string in atari)
- extension (in particular out of a ladder)
- avoid self-atari
- atari defense (in particular when there is a ko)
- distance to border (optimum distance = 3 in 19x19 Go)
- short distance to previous moves
- short distance to the move before the previous move
- locations which have probability nearly 1/3 of being of one's color at the end of the game are preferred.

The following rules are used in our 19x19 implementation, and they improve the results.

- Territory line (i.e., line number 3)
- Line of death (i.e., first line)
- Peep-connect (i.e., connect two strings when the opponent threatens to cut)
- Hane (a move which "reaches around" one or more of the opponent's stones)
- Threat

– Connect
– Wall
– Bad Kogeima(same pattern as a knight's move in chess)
– Empty triangle (three stones making a triangle without any surrounding opponent's stone)

Patterns and rules are used both (i) as an initial number of RAVE simulations, and (ii) as an additive term in H. The additive term (ii) is proportional to the number of AMAF-simulations.

The shapes that constitute the rules are illustrated in Figure 1. A naive hand tuning of parameters is performed, only for the simulations added in the AMAF statistics. They provide a 63.9±0.5 % winning rate against the version without these improvements. We are optimistic on the fact that tuning the parameters will in general strongly improve the results. Since the early developments of MoGo, some "cut" bonuses are included (i.e., advantages for playing at locations which match "cut" patterns, i.e., patterns for which a location prevents the opponent from connecting two groups).

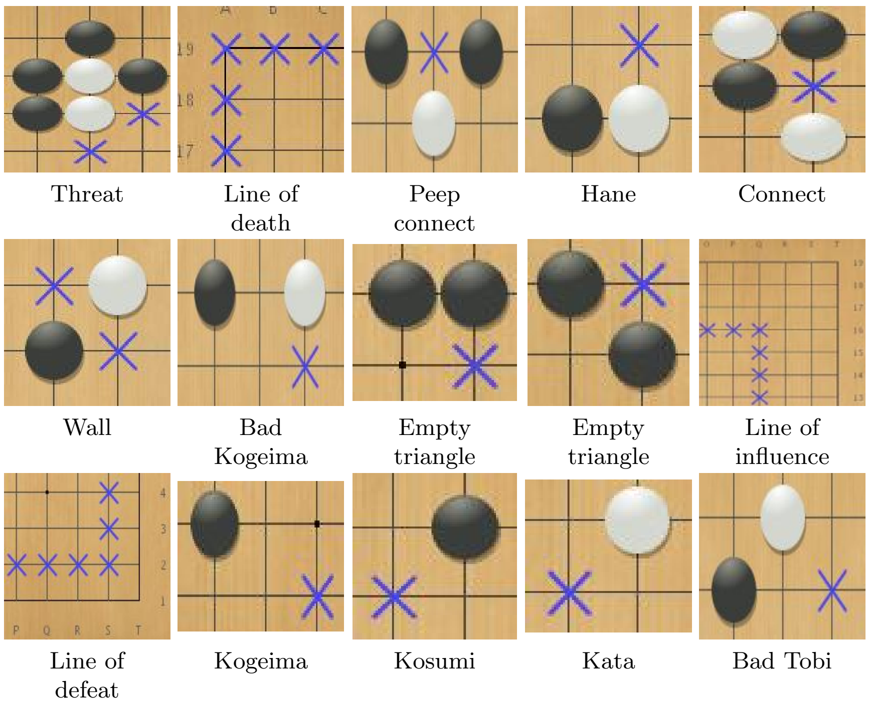


Fig. 1. Fifteen shapes that require exact matches are required for applying the bonus/malus rule. In all cases, the shapes are presented for the black player: the feature applies for a black move at one of the crosses. The reverse pattern applies for White. Threat is not an exact shape to be matched but just an example. In general, Black has a bonus for simulating one of the liberties of an enemy string with exactly two liberties, i.e., to generate an atari.

Following [9], we built a model for evaluating the probability that a move is played, conditionally to the fact that it matches some pattern. When a node is created, the pattern matching is called, and the probability it proposes is used as explained later (Eq. 1). The pattern matching is computationnally expensive and we had to tune the parameters in order to achieve positive results.

The following parameters had to be modified, when this model was included in H:

– time scales for the convergence of the weight of online statistics to 1 (see Eq. 1) are increased;
– the number of simulations of a move at a given node before the subsequent nodes is created is increased (because the computational cost of a creation is higher);
– the optimal coefficients of expert rules are modified;
– importantly, the results were greatly improved by *adding the constant* C (see Eq. 1). This is the last line of Table 1.

Results are presented in Table 1.

Table 1. Effect of adding patterns extracted from professional games in MoGo. The first tuning of parameters is the tuning of α, β, and γ as functions of $n(d)$ (see Eq. 1) and of coefficients of expert rules. A second tuning consists in tuning constant C in Eq. 1.

Tested version	Against	Conditions of games	Success rate
MoGo + patterns	MoGo without patterns	3000 sims/move	56 % ± 1%
MoGo + patterns	MoGo without patterns	2s/move	50.9 % ± 1.5 %
MoGo + patterns + tuning of coefficients	MoGo + patterns	1s/move	55.2 % ± 0.8 %
MoGo + patterns + tuning of coefficients + adding and tuning C	MoGo + patterns + tuning of coefficients	1s/move	61.72 % ± 3.1 %

3 Improving Monte-Carlo Simulations

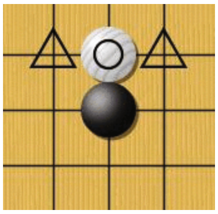
There exists no easy criterion to evaluate the effect of a modification of the Monte-Carlo simulator on the global MCTS algorithm in the case of Go (or more generally two-player games). Many people have tried to improve the MC engine by increasing its level (the strength of the Monte-Carlo simulator as a stand-alone player), but it is shown clearly in [11,4,12] that this is not a good criterion: an MC engine MC_1 which plays significantly better than another MC_2 can lead to less good results than MC_2 as a module in MCTS. Some MC engines have been learnt on datasets [10], still the results may strongly improve by changing the constants manually. In that sense, designing and calibrating an MC engine remains an open challenge: one has to experiment a modification intensively in order to validate it.

Various shapes are defined in [11,12,13,14]. Moreover, [12] uses patterns and expertise as explained in Algorithm 2. Below, we present three new improvements, two of them rely on an increased diversity when the computational power increases; in both cases, the improvement is negative or negligible for small computational power and becomes highly significant when the computational power increases. The third improvement deals with the semeai.

3.1 Fill the Board: Random Perturbations of the Monte-Carlo Simulations

The principle of the first improvement is to play first on locations of the board where there is large empty space. The idea is to increase the number of locations at which Monte-Carlo simulations can find pattern-matching in order to diversify the Monte-Carlo simulations.

As trying every position on the board would take too much time, the following procedure is used instead. A location on the board is chosen randomly; if the 8 surrounding positions are empty, the move is played, else the following $N-1$ positions on the board are tested; N is a parameter of the algorithm. This modification introduces more diversity in the simulations: it is due to the fact that the Monte-Carlo player uses a large number of patterns. When patterns match, one of them is played. So the simulations have only a few ways of playing when solely a small number of patterns match; in particular at the beginning of the game, when there are just a few stones on the goban. As this modification is played before the patterns, it leads to more diversified simulations (Figure 2 (left)). The detailed algorithm is presented in Algorithm 3, experiments in Figure 2 (right).



9x9 board		19x19 board	
Nb of playouts per move or time/move	Success rate	Nb of playouts per move or time /move	Success rate
10 000	52.9 % ± 0.5%	10000	49.3 ± 1.2 %
5s/move, 8 cores	54.3 % ± 1.2 %	5s/move, 8 cores	77.0 % ± 3.3 %
100 000	55.2 % ± 1.4 %	100 000	73.7 % ± 2.9%
200 000	55.0 % ± 1.1 %	200 000	78.4 % ± 2.9 %

Fig. 2. Left: diversity loss when the "fillboard" option was not applied: the white stone is the last move, and the black player, starting a Monte-Carlo simulation, can only play at one of the locations marked by triangles. Right: results associated to the "fillboard" modification. As the modification leads to a computational overhead, results are better for a fixed number of simulations per move; however, the improvement is clearly significant. The computational overhead is reduced when a multi-core machine is used: the concurrency for memory access is reduced when more expensive simulations are used, and therefore the difference between expensive and cheap simulations decays as the number of cores increases. This element also shows the easier parallelization of heavier playouts.

Algorithm 3. Algorithm for choosing a move in MC simulations, including the "fill board" improvement. We also experimented with a constraint of 4, 12 and 22 empty locations instead of 8, but results were disappointing.

```
if the last move is an atari, then
    Save the stones which are in atari if possible.
else
    "Fill board" part.
    for $i \in \{1, 2, 3, 4, \ldots, N\}$ do
        Randomly draw a location $x$ on the goban.
        IF $x$ is an empty location and the eight locations around $x$ are empty, play $x$
        (exit).
    end for
    End of "fill board" part.
    Sequential move, if any (see above).
    Capture move, if any (see above).
    Random legal move, if any (see above).
end if
```

3.2 The "Nakade" Problem

A known weakness of MoGo, as well as many MCTS programs, is that *nakade* is not correctly handled. We will use the term *nakade* to design a situation in which a surrounded group has a single large internal, enclosed space in which the player would not be able to establish two eyes if the opponent plays correctly.

The group is therefore dead, but the baseline Monte-Carlo simulator (Algorithm 2) sometimes estimates that it lives with a high probability, i.e., the MC simulation does not necessarily lead to the death of this group. Therefore, the tree will not grow in the direction of moves preventing difficult situations with *nakade* — MoGo just considers that this is not a dangerous situation.

This will lead to a false estimate of the probability of winning. As a consequence, the MC part (i.e., the module choosing moves for situations which are not in the tree) must be modified so that the winning probability reflects the effect of a *nakade* .

Interestingly, as most MC tools have the same weakness, and also as MoGo is mainly developed by self-play, the weakness concerning the *nakade* almost never appeared before humans found the weakness (see post from D. Fotland called "UCT and solving life and death" on the computer-Go mailing list). It would be theoretically possible to encode in MC simulations a large set of known *nakade* behaviors, but this approach has two weaknesses: (i) it is expensive and MC simulations must be very fast and (ii) abruptly changing the MC engine very often leads to unexpected disappointing effects. Therefore we designed the following modification: if a contiguous set of exactly 3 free locations is surrounded by opponent's stones, then we play at the center (the vital point) of this "hole". The new algorithm is presented in Algorithm 4.

We validate the approach by two different experiments: (i) known positions in which old MoGo does not choose the right move (Figure 3) and (ii) games confronting the new MoGo vs the old MoGo (Table 2).

Algorithm 4. New MC simulator, reducing the *nakade* problem

```
if the last move is an atari, then
    Save the stones which are in atari if possible.
else
    Beginning of the nakade modification
    for $x$ in one of the 4 empty locations around the last move do
        if $x$ is in a hole of 3 contiguous locations surrounded by enemy stones or the sides of the goban then
            Play the center of this hole (exit).
        end if
    end for
    End of the nakade modification
    "Fill board" part (see above).
    Sequential move, if any (see above).
    Capture move, if any (see above).
    Random legal move, if any (see above).
end if
```

We also show that our modification is not sufficient for all cases: in the game presented in Fig. 3 (e), MoGo lost by a poor evaluation of a nakade situation, which is not covered by our modification.

Table 2. Experimental validation of the nakade modification: modified MoGo versus baseline MoGo. Seemingly, the higher the number of simulations (which is directly related to the level), the higher the impact.

Number of simulations per move	Success rate	Number of simulations per move	Success rate
9x9 board		19x19 board	
10000	52.8 % ± 0.5%		
100000	55.6 % ± 0.6 %	100 000	53.2 % ± 1.1%
300000	56.2 % ± 0.9 %		
5s/move, 8 cores	55.8 % ± 1.4 %		
15s/move, 8 cores	60.5 % ± 1.9 %		
45s/move, 8 cores	66.2 % ± 1.4 %		

3.3 Approach Moves

Correctly handling life and death situation is a key point in improving the MC engine . Reducing the probability of simulations in which a group which should clearly live dies (or vice versa) improves the overall performance of the algorithm. For example, in Fig. 4, black should play in B before playing in A for killing A. This is an *approach move*. We implemented it as presented in algorithm 5. This modification provides a success rate of

- 52.68 % (± 0.33 %) in 9x9 with 20,000 simulations per move;
- 54.69 % (± 2.27%) in 19x19 with 50,000 simulations per move.

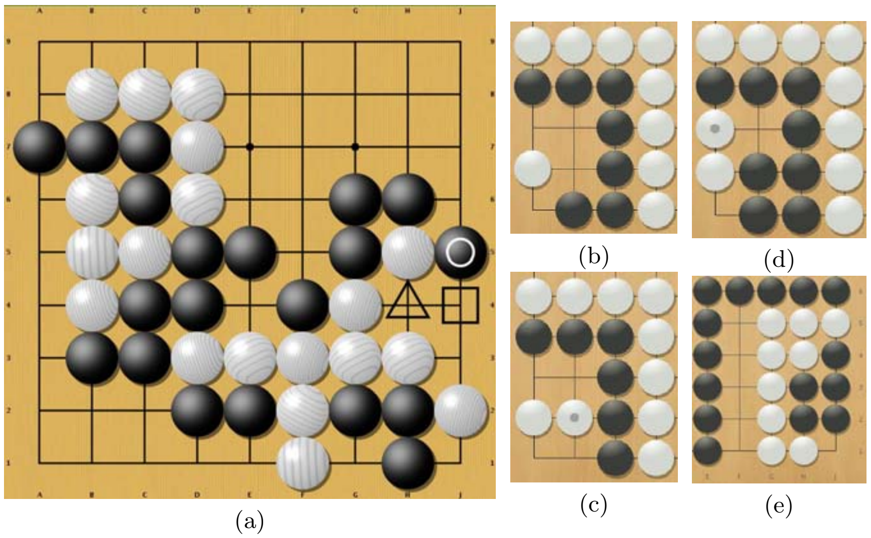


Fig. 3. In Figure (a) (a real game played and lost by MoGo), MoGo (White) without specific modification for the *nakade* chooses H4; Black plays J4 and the group F1 is dead (MoGo loses). The right move is J4; this move is chosen by MoGo after the modification presented in this section. Examples (b), (c), and (d) are other similar examples in which MoGo (as Black) evaluates the situation poorly and does not realize that his group is dead. The modification solves the problem. (e) An example of a more complicated *nakade*, which is not solved by MoGo - we have no generic tool for solving the *nakade*.

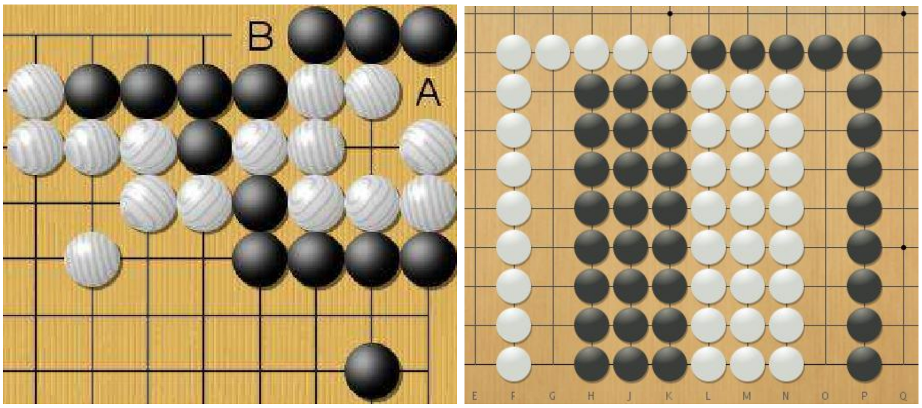


Fig. 4. Left: Example of situation which is poorly estimated without approach moves. Black should play *B* before playing *A* for killing the white group and live. Right: situation which is not handled by the "approach moves" modification.

Algorithm 5. New MC simulator, implementing approach moves. Random is a random variable uniform on $[0, 1]$.

```
if the last move is an atari, then
  Save the stones which are in atari if possible.
else
  Nakade modification (see above).
  "Fill board" part (see above).
  if there is an empty location among the 8 locations around the last move which
  matches a pattern then
    Randomly and uniformly select one of these locations.
    if this move is a self-atari and can be replaced by a a connection with another
    group and random $< 0.5$ then
      Play this connection (exit).
    else
      Play the select location (exit).
    end if
  else
    Capture move, if any (see above).
    Random legal move, if any (see above).
  end if
end if
```

We can see in Fig. 4 that some semeai situations are handled by this modification: MoGo now clearly sees that Black, playing first, can kill in Fig. 4. However, this does not solve more complicated semeais as, e.g., Fig. 4 (e).

4 Conclusions

The number of ideas that the MCTS framework offers is overwhelming. In our search for improvements we faced many negative results. However, there were positive results too. Below we condense our results to four conclusions.

First, for computers as well as for humans, all time scales of learning are important.We mention (1) offline knowledge (strategic rules and patterns) as in [2,1]; (2) online information (i.e., analysis of a sequence by mental simulations) [4]; and (3) transient information (extrapolation as a guide for exploration).

Second, reducing diversity has been a good idea in Monte-Carlo; [12] has shown that introducing several patterns and rules greatly improve the efficiency of Monte-Carlo Tree-Search. However, plenty of experiments with the aim of increasing the level of the Monte-Carlo simulator as a stand-alone player have given negative results - diversity and playing strength are too conflicting objectives. It is even not clear for us that the goal may be assumed to be a compromise between these two criteria. We can only clearly claim that increasing the diversity becomes increasingly more important as the computational power increases, as shown in Section 3.

Third, approach moves are an important feature. It makes MoGo more reasonable in some difficult situations in corners. We believe that strong improvements can arise as generalizations of this idea, for solving the prominent semeai case.

Fourth, guiding and controlling the exploration by a UCT term

$$\sqrt{\log \frac{\text{nbSims of father nodes}}{\text{nbSims of child node}}}$$

as in UCB is important, in particular when scores are naive empirical success rates. However, the optimal constant in the exploration term becomes 0 when learning is improved (at least in MoGo, and the constant is very small in several UCT-like programs also). In MoGo, the constant in front of the exploration term was not null before the introduction of RAVE values in [4]; it is now 0. A new term has provided a prevailing important improvement as an exploration term: the constant C in Eq. 1.

Acknowledgments

We thank Bruno Bouzy, Rémi Munos, Yizao Wang, Rémi Coulom, Tristan Cazenave, Jean-Yves Audibert, David Silver, Martin Mueller, KGS, Cgos, and the computer-go mailing list for many interesting discussions. Many thanks to the French Federation of Go and to Recitsproque for having organized an official game against a high-level human; many thanks also to the several players from the French and Taiwanese Federations of Go who accepted to play and discuss test games against MoGo.

References

1. Chaslot, G.M.J.B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for monte-carlo tree search. In: Wang, P., et al. (eds.) Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007), pp. 655–661. World Scientific Publishing Co. Pte. Ltd., Singapore (2007)
2. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: Ciancarini, P., van den Herik, H.J. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
3. Kocsis, L., Szepesvari, C.: Bandit-based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
4. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: ICML 2007: Proceedings of the 24th international conference on Machine learning, New York, NY, USA, pp. 273–280. ACM Press, New York (2007)
5. Brügmann, B.: Monte-Carlo Go (Unpublished) (1993)
6. Bouzy, B., Helmstetter, B.: Monte-Carlo Go developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) 10th Advances in Computer Games, pp. 159–174 (2003)
7. Coquelin, P.A., Munos, R.: Bandit algorithms for tree search. In: Proceedings of UAI 2007 (2007)

8. Gelly, S., Hoock, J.B., Rimmel, A., Teytaud, O., Kalemkarian, Y.: The parallelization of monte-carlo planning. In: Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008), pp. 198–203 (2008) (to appear)
9. Bouzy, B., Chaslot, G.M.J.B.: Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. In: Kendall, G., Lucas, S. (eds.) IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK, pp. 176–181 (2005)
10. Coulom, R.: Computing elo ratings of move patterns in the game of go. In: Computer Games Workshop, Amsterdam, The Netherlands (2007)
11. Bouzy, B., Chaslot, G.M.J.B.: Monte-Carlo Go Reinforcement Learning Experiments. In: Kendall, G., Louis, S. (eds.) IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA, pp. 187–194 (2006)
12. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii, pp. 175–182 (2007)
13. Bouzy, B.: Associating domain-dependent knowledge and Monte-Carlo approaches within a go program. In: Chen, K. (ed.) Information Sciences, Heuristic Search and Computer Game Playing IV, vol. 175, pp. 247–257 (2005)
14. Ralaivola, L., Wu, L., Baldi, P.: SVM and pattern-enriched common fate graphs for the game of Go. In: Proceedings of ESANN 2005, pp. 485–490 (2005)

A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm

Markus Enzenberger and Martin Müller

Department of Computing Science, University of Alberta
mmueller@cs.ualberta.ca

Abstract. With the recent success of Monte-Carlo tree search algorithms in Go and other games, and the increasing number of cores in standard CPUs, the efficient parallelization of the search has become an important issue. We present a new lock-free parallel algorithm for Monte-Carlo tree search which takes advantage of the memory model of the IA-32 and Intel-64 CPU architectures and intentionally ignores rare faulty updates of node values. We show that this algorithm significantly improves the scalability of the FUEGO Go program.

1 Introduction

Three topics are essential for this contribution. They are discussed below: Monte-Carlo Tree Search (in 1.1), Parallel MCTS (in 1.2), and the FUEGO Go program (in 1.3).

1.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) has proved to be a successful search method in two-player board games for which it is difficult to create a good heuristic evaluation function. In the game of Go, it was first used by the programs CRAZY STONE [1] and MOGO [2], and has led to programs that for the first time have reached human master level, especially on smaller board sizes.

In MCTS, a search tree is stored in memory and expanded incrementally. Each simulated game starts with an *in-tree phase*, in which moves are selected following a sequence of nodes from the root node and choosing a child in each node based on the current value of the child and potentially additional exploration bonuses. After a node with no children is reached, it is expanded, and the game is finished using a *playout phase* in which moves are generated by a more or less randomized move generation policy. After the playout, there is an *update phase*, in which the values of the nodes used in the game are updated by the game result, such that they store the average result of all games in which the move was chosen. The simulation process is repeated until a resource limit is exceeded, for example the number of simulations, time used, or memory.

1.2 Parallel Monte-Carlo Tree Search

MCTS has the additional advantage that it is easier to parallelize than traditional programs based on alpha-beta search [3,4,5,6,7]. The most common methods have been classified by Chaslot et al. [5] as *leaf parallelization*, *root parallelization*, and *tree parallelization*. On shared memory systems, tree parallelization is the natural method that takes full advantage of the available bandwidth for communicating game results. In this method, several threads run simulations in parallel and share a common tree in memory. Usually, a global mutex is used for protecting access and modification of the tree during the in-tree and the update phase, whereas the playout phase proceeds independently in each thread and does not require locking. How well tree parallelization scales with the number of threads depends, among other things, on the ratio of the time spent in the unlocked playout phase to the total time for a game.

Chaslot et al. [5] have shown that in their Go program MANGO, tree parallelization only scales well up to four threads. Their experiments were done in 9×9 Go, in which the playout phase is shorter and the in-tree phase longer than in 19×19 Go. They also tried a more fine-grained locking algorithm, which reduced the overhead of the global mutex at the price of increasing the node size in the tree by adding a local mutex to each node. The problem here is that, due to the selectivity of MCTS, usually a large number of nodes are shared in the in-tree move sequences of the different threads; at least one node, the root node, is always shared. Therefore using local mutexes is not necessarily an improvement. However, they showed that the addition of a *virtual loss* improved the scaling of tree parallelization in MANGO.

In messages to the Computer Go mailing list [8], Coulom reports strong results with a lock-free transposition table in CRAZY STONE. Another possible approach, also implemented in CRAZY STONE, uses *spinlocks* [8], which are a form of busy waiting. Spinlocks avoid the overhead of other locking approaches. For a relatively small number of threads, the speed of spinlocks might be comparable to the lock-free approach presented here. However, an extra variable per node is needed to hold the spinlock.

1.3 The FUEGO Go Program

FUEGO is an open-source Go program developed by the Computer Go group at the University of Alberta [9]. On the Bayes-Elo ranking of the Computer Go Server [10] from January 15 2009 (16:03 UCT for 19×19; 18:19 UCT for 9×9), it is ranked as the 3rd-strongest program ever on 9×9 with a rating of 2664 Elo. On 19×19 it is the 2nd-strongest program ever with a rating of 2290 Elo. The program is written in C++ and separated into different libraries. The MCTS implementation is in the game-independent SmartGame library and is also used by external projects for other games, such as MOHEX, a Hex program by the Games Group at the University of Alberta [11].

The main Go player in FUEGO uses full-board MCTS with a number of common enhancements to the basic MCTS algorithm. Move generation in the playout phase is similar to the one originally used by MOGO; it uses patterns, liberty and locality heuristics. In the in-tree phase, children are selected based on the Rapid Action Value Estimation (RAVE) heuristic [2], which combines the current value of a move with the average value of this move in the subtree of the corresponding node. When a node is expanded, the values and counts of new children are initialized based on a static heuristic evaluation. Parallel search is supported using tree parallelization.

2 Lock-Free Multithreaded MCTS

The basic idea of FUEGO's lock-free multithreaded MCTS algorithm is to share a tree between multiple threads without using any locks. Because of specific requirements on the memory model of the hardware platform, this lock-free mode is an optional feature of the base MCTS class in FUEGO and needs to be enabled explicitly.

2.1 Modifying the Tree Structure

The first change to make the lock-free search work is in the handling of concurrent changes to the structure of the tree. FUEGO never deletes nodes during a search; new nodes are created in a pre-allocated memory array. In the lock-free algorithm, each thread has its own memory array for creating new nodes. Only after the nodes are fully created and initialized, are they linked to the parent node. This can cause some memory overhead, because if several threads expand the same node only the children created by the last thread will be used in future simulations. It can also happen that some of the children that are lost already received value updates; these updates will be lost.

The child information of a node consists of two variables: a pointer to the first child in the array, and the number of children. To avoid that another thread sees an inconsistent state of these variables, all threads assume that the number of children is valid if the pointer to the first child is not null. Linking a parent to a new set of children requires first writing the number of children, then the pointer to the first child. The compiler is prevented from reordering the writes by declaring these variables using the C++ type qualifier *volatile*.

2.2 Updating Values

The move and RAVE values are stored in the nodes as counts and mean values. The mean values are updated using an incremental algorithm. Updating them without protection by a mutex can cause updates of the mean to be lost with or without increment of the count, as well as updates of the mean occurring without increment of the count. It could also happen that one thread reads the count and meanwhile they are written by another thread, and the first thread sees an erroneous state that exists only temporarily. In practice, these faulty updates occur with a low probability and will have only a small effect on the counts and mean values. They are intentionally ignored.

The only problematic case is if a count is zero, because the mean value is undefined if the count is zero, and this case has a special meaning at several places in the search. For example, the computation of the values for the selection of children in the in-tree phase distinguishes three cases: if the move count and RAVE count is non-zero, the value will be computed as a weighted linear combination of both mean values, if the move count is zero, only the RAVE mean is used, and if both counts are zero, a configurable constant value, the *first play urgency*, is used. To avoid this problem, all threads assume that a mean value is only valid if the corresponding count is non-zero. Updating a value requires first writing the new mean value, then the new count. Again, *volatile* is used to protect the order of writes.

2.3 Platform Requirements

There are some requirements on the memory model of the platform to make the lock-free search algorithm work. Writes of the basic types *size_t, int, float*, and *pointer* must be atomic. Writes by one thread must be seen by other threads in the same order. The IA-32 and Intel-64 CPU architectures, which are used in most modern standard computers, guarantee these assumptions. They also synchronize CPU caches after writes [12].

3 Experiments

The experiments compare how well locked and lock-free searches scale with the number of threads in FUEGO in 9×9 and 19×19 Go. The comparison is against the ideal case represented by running the singlethreaded program n times longer.

3.1 Setup

The version of FUEGO was 0.3, which was released on 17 December 2008. The hardware was an Intel Xeon E5420 2.5 GHz dual quadcore system with 8 GB main memory and a 64-bit version of the GNU/Linux operating system. On this hardware, FUEGO achieves about 11,400 simulations per second per core if a search is started on an empty 9×9 board. About 53 percent of the simulation time is spent in the playout phase. On 19×19, the program achieves 2750 simulations on an empty board and spends about 69 percent of the simulation time in playouts. The maximum tree size was set to 20,000,000 nodes. Although FUEGO implements the virtual loss enhancement [5] as an option, it is disabled by default and was not used in the experiment.

The self-play experiments were performed against a fixed opponent, the single threaded version set to 1 sec per move. Three series of runs measured the percentage of wins against the standard version using a single-threaded version with n times more time per move, as well as locked and lock-free multi threaded versions with 1 sec per move and n threads. A total of 1000 games with Chinese rules and alternating player colors were played for each data point. The opening book was disabled. The games were played using the gogui-twogtp program included in the GoGui distribution [13]. GNU Go version 3.6 [14] was used as a referee for determining the result of a game.

3.2 Results

The results of the experiment are shown in Fig. 1. The error bars in the figure correspond to one standard error.

The version using a global mutex does not scale beyond two threads on 9×9 and three on 19×19. On 9×9, this is even less than what was reported by Chaslot et al. for their program MANGO, which still showed an improvement in playing strength with up to four threads.

The lock-free version scales up to seven threads on both board sizes. The playing strength with eight threads is slightly less than with seven, although one cannot say with high confidence whether this is a real effect given the statistical error of the experiment.

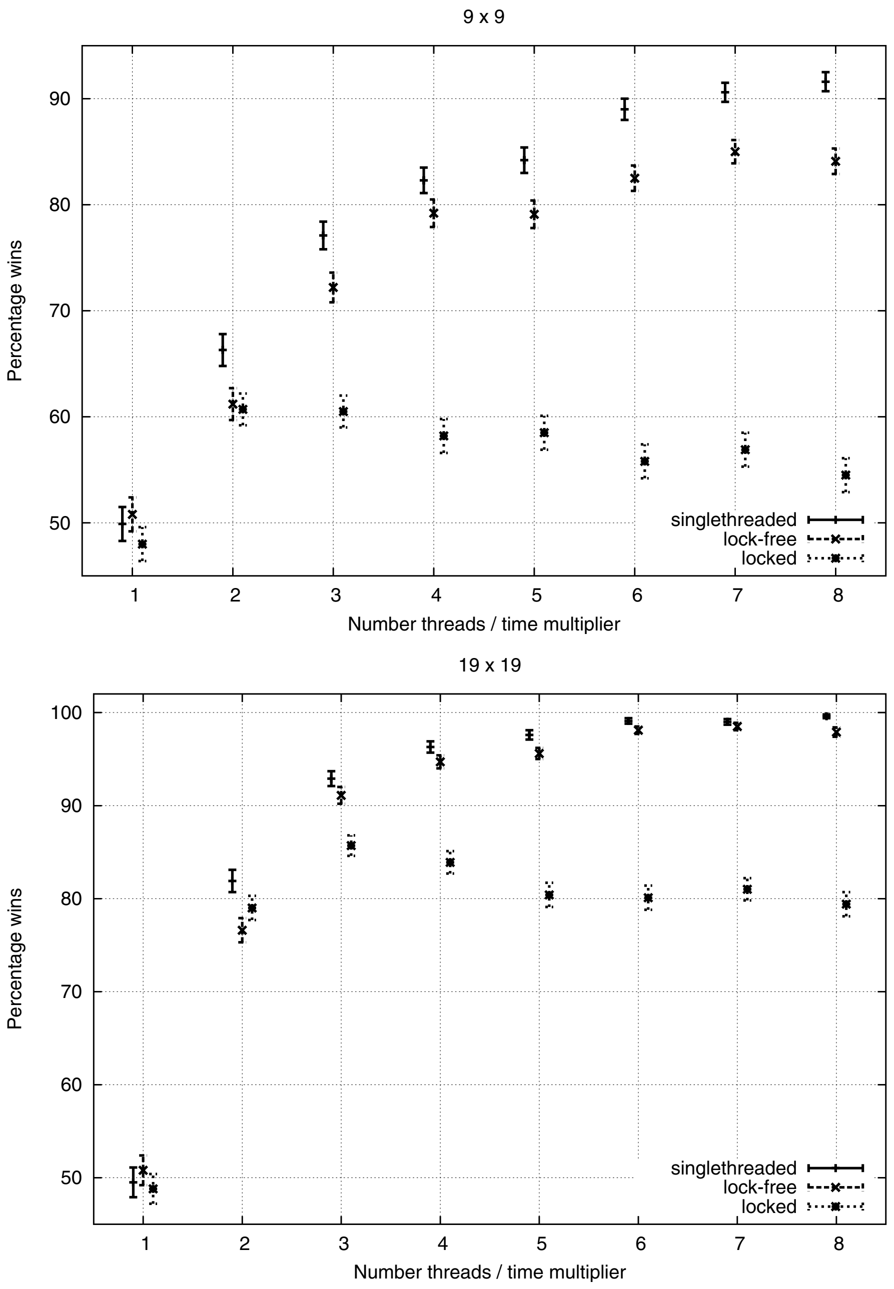


Fig. 1. Self-play performance of locked and lock-free multithreading in comparison to a single-threaded search (1 s per move)

4 Conclusion and Future Work

A lock-free multithreaded algorithm for MCTS can significantly improve the scaling of MCTS in 9×9 and 19×19 Go. Future experiments should investigate scaling with more than eight cores, and applications to other games.

Modifications of the algorithm are possible that allow it to be used on more CPU architectures. If an architecture does not guarantee that writes by one thread are seen by other threads in the same order and caches are synchronized after writes, then it might still be better to use explicit memory barriers at the places with write order dependencies than to use a global mutex for the whole in-tree and update phases.

Acknowledgements

The authors thank iCORE and NSERC for financial support.

References

1. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
2. Gelly, S.: A Contribution to Reinforcement Learning; Application to Computer-Go. PhD thesis, Université Paris-Sud (2007)
3. Cazenave, T., Jouandeau, N.: A parallel Monte-Carlo tree search algorithm. In: van den Herik, J., Xu, X., Ma, Z., Winands, M. (eds.) CG 2008. LNCS, vol. 5131, pp. 72–80. Springer, Heidelberg (2008)
4. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Computer Games Workshop, Amsterdam, pp. 93–101 (2007)
5. Chaslot, G., Winands, M., van den Herik, J.: Parallel Monte-Carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
6. Gelly, S., Hoock, J., Rimmel, A., Teytaud, O., Kalemkarian, Y.: On the parallelization of Monte-Carlo planning. In: Icinco, Madeira, Portugal, pp. 198–203 (2008)
7. Kato, H., Takeuchi, I.: Parallel Monte-Carlo Tree Search with simulation servers. In: 13th Game Programming Workshop, GPW 2008 (2008)
8. Coulom, R.: Lockless hash table and other parallel search ideas (2008), http://computer-go.org/pipermail/computer-go/2008-March/014537.html, http://computer-go.org/pipermail/computer-go/2008-March/014547.html (Date retrieved: April 28, 2009)
9. Enzenberger, M., Müller, M.: Fuego – an open-source framework for board games and Go engine based on Monte-Carlo tree search. Technical Report TR09-08, University of Alberta, Edmonton, 22 pages (2009)
10. Dailey, D.: Computer Go Server (2008), http://cgos.boardspace.net/ (Date retrieved: January 19, 2009)
11. Arneson, B., Hayward, R., Henderson, P.: Wolve 2008 wins Hex tournament. ICCA Journal 32(1), 49–54 (2009)

12. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 3A: System Programming Guide, Part 1, Order Number: 253668-029US (2008)
13. Enzenberger, M.: GoGui (2009), http://gogui.sf.net/ (Date retrieved: January 2, 2009)
14. Free Software Foundation: GNU Go (2009), http://www.gnu.org/software/gnugo/ (Date retrieved: January 2, 2009)

Monte-Carlo Tree Search in Settlers of Catan

István Szita[1], Guillaume Chaslot[1], and Pieter Spronck[2]

[1] Maastricht University, Department of Knowledge Engineering
[2] Tilburg University, Tilburg centre for Creative Computing
szityu@gmail.com, g.chaslot@micc.unimaas.nl, p.spronck@uvt.nl

Abstract. Games are considered important benchmark opportunities for artificial intelligence research. Modern strategic board games can typically be played by three or more people, which makes them suitable test beds for investigating multi-player strategic decision making. Monte-Carlo Tree Search (MCTS) is a recently published family of algorithms that achieved successful results with classical, two-player, perfect-information games such as Go. In this paper we apply MCTS to the multi-player, non-deterministic board game Settlers of Catan. We implemented an agent that is able to play against computer-controlled and human players. We show that MCTS can be adapted successfully to multi-agent environments, and present two approaches of providing the agent with a limited amount of domain knowledge. Our results show that the agent has a considerable playing strength when compared to game implementation with existing heuristics. So, we may conclude that MCTS is a suitable tool for achieving a strong Settlers of Catan player.

1 Motivation

General consensus states that a learning agent must be situated in an experience-rich, complex environment for the emergence of intelligence [1,2]. In this respect, games (including the diverse set of board games, card games, and modern computer games) are considered to be ideal test environments for AI research [3,4,5,6]. This is especially true when we take into account the role of games in human societies: it is generally believed that games are tools both for children and adults for understanding the world and for developing their intelligence [7,8].

Most games are abstract environments, intended to be interesting and challenging for human intelligence. Abstraction makes games easier to analyze than real-life environments, and usually provides a well-defined measure of performance. Nevertheless, in most cases, the complexity is high enough to make them appealing to human intelligence. Games are good indicators and often-used benchmarks of AI performance: Chess, Checkers, Backgammon, Poker, and Go all define important cornerstones of the development of artificial intelligence [9,10].

Modern strategic board games (sometimes called "eurogames") are increasing in popularity since their (re)birth in the 1990s. The game Settlers of Catan can be considered an archetypical member of the genre. Strategic board games are of particular interest to AI researchers because they provide a direct link between classic (two-player, perfect information) board games and video games. On the one hand, state variables of most modern board games are discrete, and decision making is turn-based. On the other hand, the gameplay in modern board games often incorporates elements of randomness, hidden information, multiple players, and a variable initial setup, which make it hard to use classical techniques such as alpha-beta pruning [11] or opening books.

Several computer implementations of Settlers of Catan exist, which typically feature a hand-designed, rule-based AI. The strength of these AIs varies, but an experienced player can defeat them easily. Few research papers are available on autonomous learning in Settlers of Catan [12], and according to the results reported therein, they are far from reaching human-level play yet. In this paper, we investigate whether it is possible to use one of the AI tools of classical board games, namely Monte-Carlo Tree Search, effectively for implementing game-playing agents for games such as Settlers of Catan.

2 The Game: Settlers of Catan

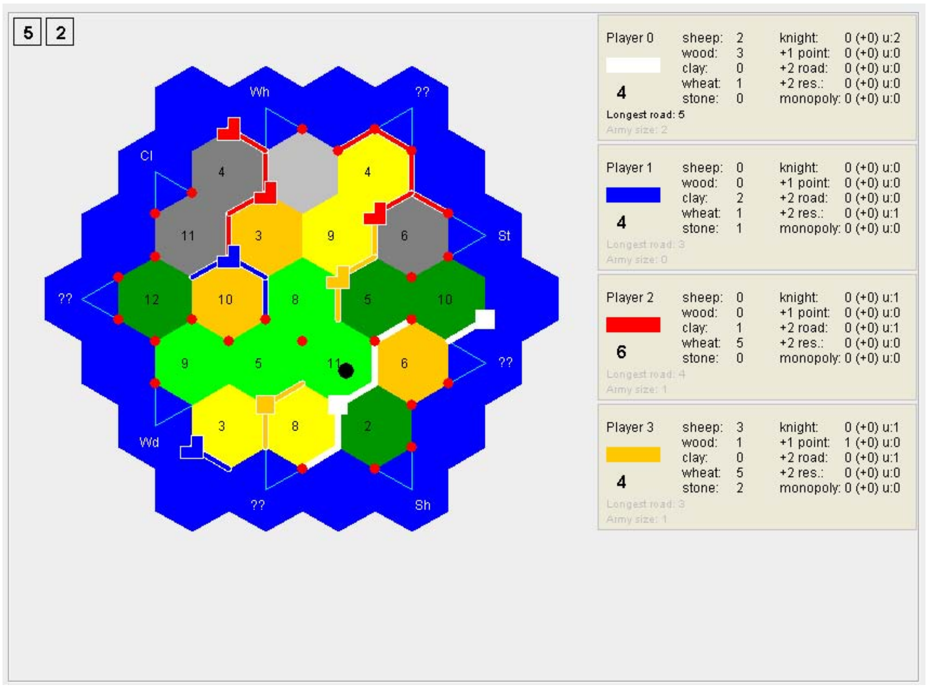
Settlers of Catan, designed by Klaus Teuber, was first published in 1995. The game achieved a huge success: it received the "Game of the Year" award of the German game critics, and it was the first "eurogame" to become widely popular outside Germany, selling more than 11 million copies, inspiring numerous extensions, successors, and computer game versions.

2.1 Game Rules

In Settlers of Catan, the players take the role of settlers inhabiting an island. They collect resources by suitable positioning of their settlements. They use these resources to build new settlements, cities, roads, and developments. The goal of the game is to be the first player who gains at least 10 *victory points*. Detailed descriptions of the rules can be easily found on the internet. Below, we summarize the rules (omitting many details).

Game board and resources. The game board representing the island is randomly assembled from 19 hexagonal *tiles* forming a large hexagon (Figure 1 displays an example game board in our SmartSettlers program). Each tile represents a field that provides one of the five types of *resources*: *wood*, *clay*, *sheep*, *wheat* or *ore*. There is also one *desert* which does not produce anything. Each non-desert tile has a *production number*. Several of the sea tiles surrounding the island contain a *port*. There is a *robber* on the island, initially residing in the desert. During the game, the players can build *settlements* and *cities* on the vertices of the hexagons, and *roads* on the edges. Settlements can be placed on any free vertex that respects the *distance rule*: no settlement may be located on a vertex that is connected to another settlement or city by exactly one edge. Players start the game with two settlements and two roads already built.

Production. Each player's turn starts by the rolling of two dice, which determines production. Any field that bears a production number equal to the sum



```
Player 0      sheep:  2     knight:    0 (+0) u:2
              wood:   3     +1 point:  0 (+0) u:0
              clay:   0     +2 road:   0 (+0) u:0
   4          wheat:  1     +2 res.:   0 (+0) u:0
              stone:  0     monopoly:  0 (+0) u:0
Longest road: 5
Army size: 2

Player 1      sheep:  0     knight:    0 (+0) u:0
              wood:   0     +1 point:  0 (+0) u:0
              clay:   2     +2 road:   0 (+0) u:0
   4          wheat:  1     +2 res.:   0 (+0) u:1
              stone:  1     monopoly:  0 (+0) u:0
Longest road: 3
Army size: 0

Player 2      sheep:  0     knight:    0 (+0) u:1
              wood:   0     +1 point:  0 (+0) u:0
              clay:   1     +2 road:   0 (+0) u:1
   6          wheat:  5     +2 res.:   0 (+0) u:0
              stone:  0     monopoly:  0 (+0) u:0
Longest road: 4
Army size: 1

Player 3      sheep:  3     knight:    0 (+0) u:1
              wood:   1     +1 point:  1 (+0) u:0
              clay:   0     +2 road:   0 (+0) u:1
   4          wheat:  5     +2 res.:   0 (+0) u:0
              stone:  2     monopoly:  0 (+0) u:0
Longest road: 3
Army size: 1
```

Fig. 1. A game state of Settlers of Catan represented in our program SmartSettlers

of the dice, produces resources in that turn. Any player (i.e., not only the player whose turn it is) who has a settlement adjacent to a producing field, receives one of the corresponding resources. For any adjacent city, he receives two. On a dice roll of 7, nothing is produced but the robber is activated. Any player who has more than seven resource cards must discard half of them. After that, the current player moves the robber to a new field. The field with the robber is *blocked*, i.e., it does not produce anything, even if its number is rolled. Furthermore, the current player can draw a random resource from one of his[1] opponents who has a field adjacent to the robber's field.

Buildings, roads, and developments. After the production phase, a player can take zero or more actions. He can use these actions to construct new buildings and roads. A new road costs *1 wood and 1 clay*, and must be connected to the player's existing road network. A new settlement costs *1 wood, 1 clay, 1 sheep, and 1 wheat*, and it must be placed in a way that it is connected to the player's road network, respecting the distance rule. Players may upgrade their settlements to cities for *3 ore and 2 wheat*. Players can also purchase a random *development card* for *1 sheep, 1 wheat, and 1 ore*. Cards can give three kinds of bonuses. (1) Some cards give 1 victory point. (2) Some cards are *Knight cards*, which can be used to activate the robber immediately. (3) The third group of

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

cards gives miscellaneous rewards such as free resources or free roads. In his turn, a player may use at most one development card.

Trading. The current player is allowed to trade resources with his opponents, as long as all involved in the trade agree. Players may also “trade with the bank”: which means that they exchange four resources of one kind for one of a different kind. If they have built a settlement or city connecting to a port, they can trade with the bank at a better ratio, depending on the kind of port.

Victory points and trophies. Next to the development cards that provide *1 victory point*, a player gains victory points for settlements, cities, and trophies. Each settlement is worth *1 victory point*, and each city *2 victory points*. The first player to use three Knight cards obtains the trophy *“Largest army”*. Subsequently, if another player uses more Knight cards, he becomes the trophy holder. Similarly, the player who is first to build a chain of five or more connected roads, obtains the trophy *“Longest road”*, and owns it until someone builds a longer chain. Both trophies are worth *2 victory points*. The game ends as soon as a player reaches at least 10 victory points. That player wins the game.

2.2 Rule Changes

From the rules it follows that Settlers of Catan has a variable initial setup, non-deterministic elements (the dice rolls), elements of imperfect information (buying of development cards, and a lack of knowledge of cards stolen), and more than two players. For an easier implementation of the game, we changed the rules to remove the elements of imperfect information. In our opinion, this change does not alter gameplay in a significant way: knowing the opponents’ development cards does not significantly alter the strategy to be followed, and information on the few cards stolen is usually quickly revealed anyway. We also chose to not let our game-playing agent initiate trades with or accept trades from other players (although it may trade with the bank). Note that the other players may trade between themselves, which handicaps our agent slightly. While we believe that these changes do not alter the game significantly, it is our intention to upgrade our implementation to conform completely to the rules of Settlers of Catan in future work.

2.3 Previous Computer Implementations

There are about ten computer implementations of Settlers of Catan available. We mention two of the strongest ones. The first is Castle Hill Studios’ version of the game, currently part of Microsoft’s MSN Games. The game features strong AI players who use trading extensively. The second is Robert S. Thomas’ JSettlers, which is an open-source Java version of the game, also having AI players. The latter is the basis of many Settlers of Catan game servers on the Internet. The JSettlers architecture is described in detail by Thomas [13]. Pfeiffer [12] also used the JSettlers environment to implement a learning agent. His agent uses hand-coded high-level heuristics with low-level model trees constructed by reinforcement learning.

3 Implementation

We implemented the Settlers of Catan game in a Java software module named *SmartSettlers*. JSettlers was used for providing a graphical user interface (GUI) and the baseline AI. Because of the Internet-based architecture of JSettlers, gameplay is fairly slow: a single four-player game takes about 10 minutes on an average PC. The learning algorithm that is investigated in our project requires thousands of simulated games before making a single step, so the JSettlers environment in itself is clearly inadequate for that purpose. Therefore, we implemented SmartSettlers as a *standalone* Java software module. It was designed for fast gameplay, move generation, and evaluation. For optimum speed, game data was divided into two parts: information that is constant throughout a game was stored in a static object, while game-state dependent information was stored in unstructured arrays of integers. The latter structure enabled quick accessing and copying of information. On an average PC, SmartSettlers is able to play around 300 games per second with randomly generated moves. JSettlers handles the SmartSettlers AI as a separate thread, querying it for decisions through a translation interface. In addition, SmartSettlers is able to run as a stand-alone program for investigating the self-play of the SmartSettlers AI. The goal of the current research is to create a strong AI agent for Settlers of Catan.

4 Monte-Carlo Tree Search

In recent years, several Monte-Carlo based techniques emerged in the field of computer games. They have already been applied successfully to many games, including Poker [14] and Scrabble [15]. Monte-Carlo Tree Search (MCTS) [16], a Monte-Carlo simulation based technique that was first established in 2006, is implemented in top-rated Go programs [17,18,19]. The algorithm is a best-first search technique which uses stochastic simulations. MCTS can be applied to any two-player, deterministic, perfect information game of finite length (but its extension to multiple players and non-deterministic games is straightforward). Its basis is the simulation of games where both the AI-controlled player and its opponents play random moves. From a single random game (where every player selects his actions randomly), very little can be learned. But from simulating a multitude of random games, a suitable strategy can be inferred. In subsection 4.1 we discuss the effect of a starting position, or otherwise stated the effect of the seating order Studies of MCTS in Go have shown that inclusion of domain knowledge can improve the performance significantly [17,20]. There are at least two possible ways to include domain knowledge: (1) using non-uniform sampling in the Monte-Carlo simulation phase and (2) modifying the statistics stored in the game tree. The approaches are discussed in Subsection 4.2 and 4.3, respectively. For a detailed description of MCTS we refer the reader to one of the previously mentioned sources.

4.1 Effect of Starting Position

Reviews of Settlers of Catan suggest that the seating order of players has a significant effect on their final rankings (although opinions differ which seat gives the best strategic position). We conducted two preliminary sets of experiments to confirm this effect. We recorded the games of four identical agents playing random moves. In our first set of experiments, all agents made random (but legal) moves, while for the second set, the agents used MCTS with 1000 simulated games per move. The score distributions for different seating orders are shown in the Figures 2 and 3. In both cases, 400 games were played.

The results show that the effect of the seating order is present and is statistically significant. It is surprising, however, that the actual effect can differ for different strategies, despite the fact that both tested strategies are rather weak. For random agents, the starting player has a distinct advantage, while for MCTS agents player 1 is worst off, and players 2 and 3 have an advantage. A possible explanation is that, for purely random players, the first player has an advantage because in general he can play one extra move in comparison to the other players. Conversely, when players follow some kind of strategy, it seems that being second or third in the seating order gives a strategic advantage. This effect is probably related to the placement of initial settlements, and it would be interesting to study the seating-order effect for stronger MCTS players that play 10,000 or more simulated games per move.

The two preliminary experiments showed that the seating order effect can introduce an unknown bias to the performances of agents. In order to eliminate this bias, the seating order was randomized for all subsequent experiments where different agents were compared.

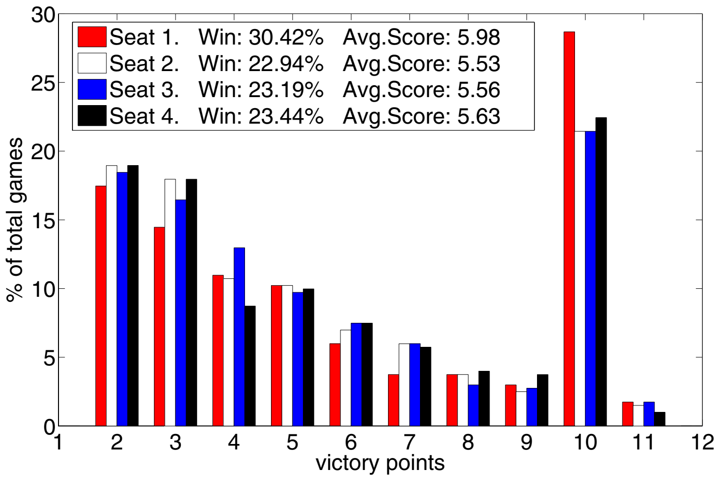


Fig. 2. The effect of seating order on the score distribution of random agents. Agents select a random legal move each turn.

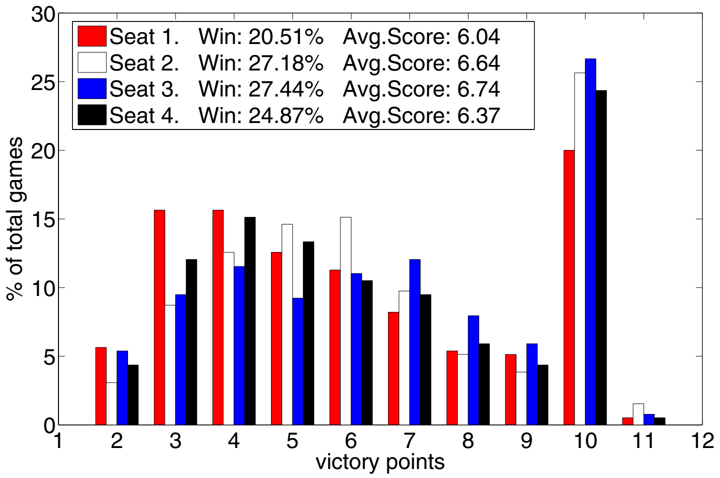


Fig. 3. The effect of seating order on the score distribution of SmartSettlers agents. Agents play 1000 simulated games per each real move.

4.2 Domain Knowledge in Monte-Carlo Simulations

If all legal actions are selected with equal probability, then the strategy played is often weak, and the quality of a Monte-Carlo simulation suffers as a result. We can use heuristic knowledge to give larger weights to actions that look promising. However, we must be careful if the selection strategy is too deterministic, then the exploration of the search space becomes too selective, and the quality of the Monte-Carlo simulation suffers. Hence, a balance between exploration and exploitation must be found in the simulation strategy.

We set the following weights for actions:

- the basic weight of each action is +1
- building a city or a settlement: +10,000; building a city or a settlement is always a good move: it gives +1 point, and also increases the resource income; all possible city/settlement-building possibilities are given the same weight; it is left to the MC simulation to differentiate between them;
- building a road: define the road-to-settlement ratio as

$$R := \frac{\text{No. of player's roads}}{\text{No. of player's settlements + cities}}$$

  and the weight of a road-building move as $10/10^R$. If there are relatively few roads, then road building should have a high weight; conversely, if there are many roads, then road building is less favourable than ending the turn;
- playing a knight card: +100 if the robber is blocking one of the player's own fields, otherwise +1;
- playing a development card: +10.

Probabilities of choosing an action in the Monte-Carlo simulation were proportional to their weights. The settings seem reasonable according to our experience with the game and with expert advice on Monte-Carlo simulations. However, the actual performance of our agent dropped significantly when using the modified probabilities instead of uniform sampling. A possible explanation is that with the given weights, the agents are too eager to build settlements and cities, while there may be a strategic advantage in giving preference to extending the roads network to reach a better strategic position. Identifying the precise reasons for the observed performance drop and working out a better set of weights (possibly in an automated process) is part of our future work.

4.3 Domain Knowledge in MCTS

Recent work [21] showed that domain knowledge can be easily injected into the tree-search aspects of MCTS, too. We can give "virtual wins" to preferred actions (and we could also give "virtual losses" to non-preferred ones). We added a quite limited amount of domain knowledge: all settlement-building actions receive 20 virtual wins, and all city-building actions receive 10. Other actions do not receive any virtual wins. This means that for each time a settlement-building action is added to the search tree, its counters for the number of visits and the number of wins are both initialized to 20.

Note that virtual wins are not propagated back to parent nodes, as that would seriously distort selection. For example, consider a situation where the player has two options: he can either build a settlement in one of 5 possible places (giving +20 virtual wins) or buy a development card first (giving no virtual wins), and build a settlement afterwards (giving +20 virtual wins for any of the 5 possible placements). If virtual wins are backpropagated, then the card-buying action obtains a huge +100 bonus. As an effect, the *potential to build* a settlement will be rated higher than the *actual building* of that settlement. This is an effect that should be avoided.

The small addition of virtual wins increased the playing strength of our agent considerably, so subsequent tests were run with this modification. Further optimization of distributing virtual wins should be possible, and is part of our plans for future work.

5 Playing Strength

We tested the playing strength of SmartSettlers against the JSettlers AIs (5.1) and against humans (5.2).

5.1 Testing MCTS against JSettlers

Against the baseline heuristics of JSettlers we tested three different AIs: the random player, MCTS with $N = 1000$ simulated games per move, and MCTS with $N = 10000$. In all experiments, three JSettlers agents were playing against one other AI. For each experiment, 100 games were played. Following our preliminary investigations, we assume that no biasing terms are present, so results for the three JSettlers agents are interchangeable. Therefore we obtain three times as many data points for the JSettlers agents than for the other AIs. The winning percentages presented below correspond to a *single* JSettlers agent. This means that a player that is equal in strength to JSettlers wins 25% of the time. The results are shown in Figure 4.

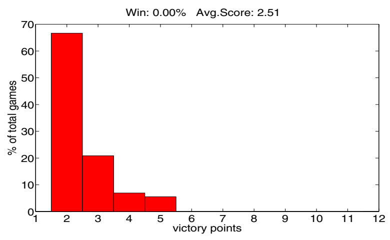
From Figure 4 (a) and (b) we may conclude that, as expected, the random player is very weak: it does not win a single game, and in fact, in most games it does not even score a point (besides the points it receives for its initial settlements). From Figure 4 (c) and (d) we may conclude that MCTS with 1000 simulated games is roughly as strong as JSettlers: it wins 27% of the games, and the score distribution is also similar. Finally, from Figure 4 (e) and (f) we may conclude that MCTS with 10000 simulated games is convincingly better than JSettlers: it wins 49% of all games, and reaches good scores even when it does not win.

5.2 Testing MCTS against Humans

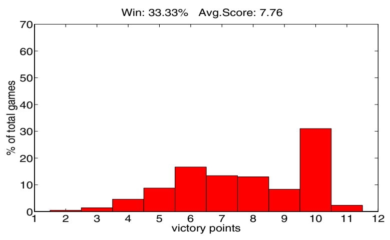
To test how SmartSettlers performs against humans, the first author, who is an accomplished Settlers of Catan player, played a few dozen games against a combination of two JSettlers' agents and one SmartSettlers agent. While the number of games played is not sufficient for drawing statistically relevant conclusions, these games do provide some information about the playing strength and style of our agent. Qualitatively speaking, we assess that the SmartSettlers agent makes justifiable moves that often coincide with moves that a human would play. Still, we found that an expert human player can confidently beat the SmartSettlers agent.

For an analysis of the possible reasons for the human supremacy over SmartSettlers, we examined different strategies for Settlers of Catan. There are two major "pure" winning strategies (in practice, human players often follow a combination of these two): the "ore-and-wheat" strategy and the "wood-and-clay" strategy. The former focuses on building cities and buying development cards, Knight cards (receiving +2 points for the largest army), and 1-point cards. The latter strategy focuses on building settlements and an extensive road network (which may lead to receiving +2 points for the longest road). Our SmartSettlers agent seems to prefer the "ore-and-wheat" strategy, together with building as many roads as possible, but not building as many settlements as an expert human player would.

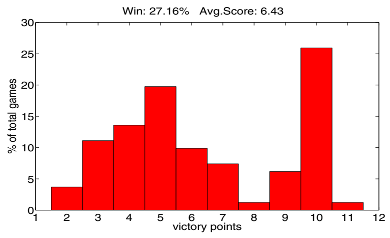
The source of this behavior is probably that MCTS does not look forward in the game tree to a sufficient depth. Building a settlement requires four different resources. Collecting these often requires 3 to 4 rounds of waiting, trading, and luck. In the meantime, the agent can spend some of the resources to buy a road to obtain a marginal advantage. While this may be much less preferable in the long run (for example, obtaining 2 victory points for the longest road is a huge advantage in the short run, but it does not help increasing the production rates), the agent may not see this, as it does not analyze properly the game tree at that depth. Increasing the number of simulated games per turn would



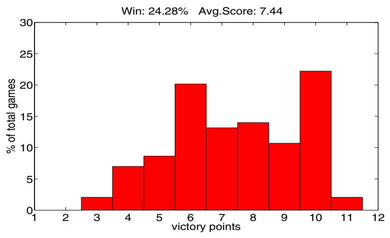
(a) Random player



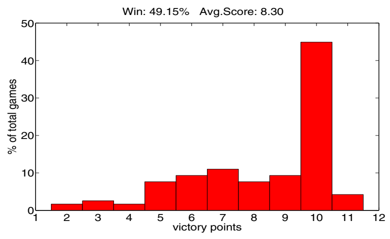
(b) JSettlers player



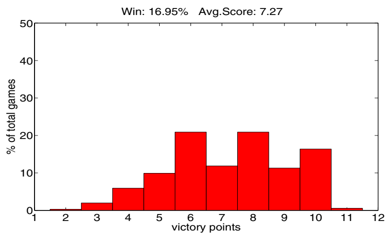
(c) MCTS-1000 player



(d) JSettlers player



(e) MCTS-10000 player



(f) JSettlers player

Fig. 4. Three sets of score distributions. Top: (a) Random player vs. (b) JSettlers. The random player won 0% of the time. Middle: (c) MCTS player ($N = 1000$) vs. (d) JSettlers. The MCTS-1000 player won 27% of the time. Bottom: (e) MCTS player ($N = 10,000$) vs. (f) JSettlers. The MCTS-10,000 player won 49% of the time.

probably alleviate this weakness, but at the cost of a significant speed decrease. An alternative approach we wish to pursue is to improve the move-selection heuristics for the Monte-Carlo simulations.

6 Future Work

Our plans for future work can be grouped into two categories. First, we plan to update our software so that it complies fully with the original rules of Settlers of Catan. To this end, we need to implement trading (which should be straightforward) and the handling of hidden information. The latter requires the implementation of a plain inference routine (keeping track of the possible values of the unknown cards) and an extension of MCTS to the case where the current state has uncertainty.

Second, we believe that the playing strength of our agent can be improved considerably by injecting domain knowledge in a proper way. There are at least two opportunities to place domain knowledge: by modifying the heuristic action-selection procedure inside the MCTS tree (by adding virtual wins to encourage favorable actions), and by modifying the move-selection probabilities in Monte-Carlo move selection. In both cases, it is possible to extract the necessary domain knowledge from a large database of played games.

7 Conclusions

In this paper we described an agent that learns to play Settlers of Catan. For move selection, the agent applies Monte-Carlo Tree Search, augmented with a limited amount of domain knowledge. The playing strength of our agent is notable: it convincingly defeats the hand-coded AI of JSettlers, and is a reasonably strong opponent for humans.

So far, applications of MCTS have been mainly constrained to Go. The success of our Settlers of Catan agent indicates that MCTS may be a viable approach for other multi-player games with complex rules.

Acknowledgments

The authors are supported by grants from the Dutch Organisation for Scientific Research (NWO grant 612.066.406 for the first and third author, NWO grant 612.066.409 for the second author).

References

1. Grabinger, R., Dunlap, J.: Rich environments for active learning: a definition. Association for Learning Technology Journal 3(2), 5–34 (1995)
2. Singh, S.P., Barto, A.G., Chentanez, N.: Intrinsically motivated reinforcement learning. In: Advances in Neural Information Processing Systems, vol. 17 (2005)
3. Dekker, S., van den Herik, H., Herschberg, I.: Perfect knowledge revisited. Artificial Intelligence 43(1), 111–123 (1990)
4. Laird, J., van Lent, M.: Human-level AI's killer application: Interactive computer games. AI Magazine 22(2), 15–26 (2001)
5. Sawyer, B.: Serious games: Improving public policy through game-based learning and simulation. Foresight and Governance Project, Woodrow Wilson International Center for Scholars Publication 1 (2002)
6. Schaeffer, J., van den Herik, H.: Games, computers, and artificial intelligence. Artificial Intelligence 134, 1–7 (2002)

7. Caldera, Y., Culp, A., O'Brien, M., Truglio, R., Alvarez, M., Huston, A.: Children's play preferences, construction play with blocks, and visual-spatial skills: Are they related? International Journal of Behavioral Development 23(4), 855–872 (1999)
8. Huitt, W.: Cognitive development: Applications. Educational Psychology Interactive (1997)
9. van den Herik, H., Iida, H. (eds.): Games in AI Research, Van Spijk, Venlo, The Netherlands (2000)
10. van den Herik, H.J., Uiterwijk, J.W.H.M., van Rijswijck, J.: Games solved: Now and in the future. Artificial Intelligence 134, 277–311 (2002)
11. Marsland, T.A.: Computer chess methods. In: Shapiro, S. (ed.) Encyclopedia of Artificial Intelligence, pp. 157–171. J. Wiley & Sons, Chichester (1987)
12. Pfeiffer, M.: Reinforcement learning of strategies for settlers of catan. In: Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education (2004)
13. Thomas, R.: Real-time Decision Making for Adversarial Environments Using a Plan-based Heuristic. PhD thesis, Northwestern University, Evanston, Illinois (2003)
14. Billings, D., Davidson, A., Schaeffer, J., Szafron, D.: The challenge of poker. Artificial Intelligence 134(1), 201–240 (2002)
15. Sheppard, B.: World-championship-caliber scrabble. Artificial Intelligence 134(1), 241–275 (2002)
16. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
17. Chaslot, G., Saito, J., Bouzy, B., Uiterwijk, J., van den Herik, H.: Monte-carlo strategies for computer go. In: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, pp. 83–90 (2006)
18. Chaslot, G., Winands, M., van den Herik, H., Uiterwijk, J., Bouzy, B.: Progressive strategies for monte-carlo tree search. New Mathematics and Natural Computation 4(3), 343 (2008)
19. Gelly, S., Wang, Y.: Exploration exploitation in go: UCT for monte-carlo go. In: NIPS-2006: On-line trading of Exploration and Exploitation Workshop (2006)
20. Bouzy, B., Chaslot, G.: Monte-Carlo go reinforcement learning experiments. In: IEEE 2006 Symposium on Computational Intelligence in Games, pp. 187–194 (2006)
21. Chatriot, L., Gelly, S., Jean-Baptiste, H., Perez, J., Rimmel, A., Teytaud, O.: Including expert knowledge in bandit-based monte-carlo planning, with application to computer-go. In: Girgin, S., Loth, M., Munos, R., Preux, P., Ryabko, D. (eds.) EWRL 2008. LNCS (LNAI), vol. 5323. Springer, Heidelberg (2008)

Evaluation Function Based Monte-Carlo LOA

Mark H.M. Winands[1] and Yngvi Björnsson[2]

[1] Games and AI Group, Department of Knowledge Engineering,
Faculty of Humanities and Sciences,
Maastricht University, Maastricht, The Netherlands
m.winands@maastrichtuniversity.nl

[2] School of Computer Science, Reykjavík University, Reykjavík, Iceland
yngvi@ru.is

Abstract. Recently, Monte-Carlo Tree Search (MCTS) has advanced the field of computer Go substantially. Also in the game of Lines of Action (LOA), which has been dominated so far by $\alpha\beta$, MCTS is making an inroad. In this paper we investigate how to use a positional evaluation function in a Monte-Carlo simulation-based LOA program (MC-LOA). Four different simulation strategies are designed, called Evaluation Cut-Off, Corrective, Greedy, and Mixed. They use an evaluation function in several ways. Experimental results reveal that the Mixed strategy is the best among them. This strategy draws the moves randomly based on their transition probabilities in the first part of a simulation, but selects them based on their evaluation score in the second part of a simulation. Using this simulation strategy the MC-LOA program plays at the same level as the $\alpha\beta$ program MIA, the best LOA-playing entity in the world.

1 Introduction

The $\alpha\beta$ search algorithm has for decades been the standard technique used by game programs for playing two-person zero-sum games, such as chess and checkers (and many others). Over the years, many search enhancements have been proposed to further improve its effectiveness. However, for some games this approach has been less successful, either because of a large branching factor preventing a deep look-a-head, or because of complications in constructing an effective evaluation function.

In recent years a new paradigm for game-tree search has emerged, the so-called Monte-Carlo Tree Search (MCTS) [9,12]. In the context of game playing, Monte-Carlo simulations were first used as a mechanism for dynamically evaluating the merits of leaf nodes of a traditional $\alpha\beta$-based search [1,3,4], but under the new paradigm MCTS has evolved into a full-fledged best-first search procedure that replaces traditional $\alpha\beta$-based search altogether. MCTS has in the past couple of years substantially advanced the state-of-the-art in several game domains where $\alpha\beta$-based search has had difficulties. In particular we mention computer Go, but other domains include General Game Playing [10], Phantom Go [5], and Amazons [13]. These are, however, all examples of game domains where either a large branching factor or a complex static state evaluation do restrain $\alpha\beta$ search in one way or another.

In this paper we introduce an improved MCTS variant that performs at the same level as a world-class $\alpha\beta$-based player in the game Lines of Action (LOA) [14]. This is an important milestone for MCTS, because up until now the traditional game-tree search approach has generally been considered to be better suited for LOA, which features both a moderate branching factor and good state evaluators (the best LOA programs use highly sophisticated evaluation functions). The previously best game-playing programs for this game, MIA [16], Bing, YL [2], and Mona [2], are all $\alpha\beta$ based. Recent work on using a special MCTS-solver variant in the world-class LOA program MIA did improve the program's tactical ability, although it still lacked the overall robustness to play against its $\alpha\beta$-based counterpart on a close to equal footing [18]. In this work our MCTS-solver variant enriched with a positional evaluation function, is able to hold its own. Moreover, the solver is easily parallelizable and when allowed to use more than one processor it does handily outperform the best $\alpha\beta$-based LOA programs.

The article is organized as follows. Section 2 explains briefly the rules of LOA. In Sect. 3 we discuss the application of MCTS in the game of LOA. In Sect. 4 we present several play-out strategies for MC-LOA. We test them in Sect. 5. Finally, Sect. 6 gives conclusions and an outlook on future research.

2 Lines of Action

Lines of Action (LOA) is a two-person zero-sum connection game with perfect information. It is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The black pieces are placed along the top and bottom rows of the board, while the white pieces are placed in the left- and right-most files of the board (see Fig. 1a). The players alternately move a piece, starting with Black. A piece moves in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement (see Fig. 1b). A player may jump over its own pieces, but not over the opponent's. The rule is that opponent's pieces are captured (and removed from the board) by landing on them. The goal of the players is to be the first to create a configuration on the board in which all own pieces are connected in one unit (e.g., see Fig. 1c). The connections within the unit may be either orthogonal or diagonal. In the case of simultaneous connection, the game is drawn.

3 Monte-Carlo LOA

In this section we discuss how we applied MCTS in LOA. First, we briefly sketch MCTS and its variant MCTS-Solver in Subsect. 3.1. Next, we explain MCTS (-Solver) in detail in Subsect. 3.2. Finally, we explain how we parallelized the search in Subsect. 3.3.

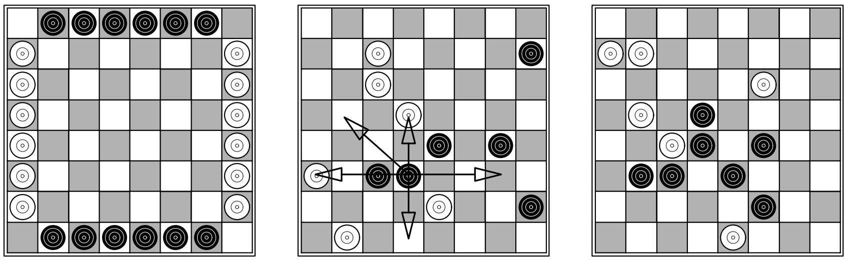


Fig. 1. (a) The initial position. (b) Example of possible moves. (c) A terminal position.

3.1 MCTS and MCTS-Solver

Monte-Carlo Tree Search (MCTS) [9,12] is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space. Using the results of previous explorations, the algorithm gradually builds up a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic steps, repeated as long as there is deliberation time left. The steps are as follows. (1) In the *selection step* the tree is traversed from the root node until we reach a node E, where we select a position that is not added to the tree yet. (2) Next, during the *play-out step* moves are played in self-play until the end of the game is reached. The result R of this "simulated" game is $+1$ in case of a win for Black (the first player in LOA), 0 in case of a draw, and -1 in case of a win for White. (3) Subsequently, in the *expansion step* children of E are added to the tree. (4) Finally, R is propagated back along the path from E to the root node in the *backpropagation step*. When time is up, the move played by the program is the child of the root with the highest value.

MCTS is unable to *prove* the game-theoretic value. However, in the long run MCTS equipped with the UCT formula [12] *converges* to the game-theoretic value. For a fixed termination game such as Go, MCTS is able to find the optimal move relatively fast in endgame positions [20]. But in a sudden-death game such as LOA, where the main line towards the winning position is narrow, MCTS may often lead to an erroneous outcome because the nodes' values in the tree do not converge fast enough to their game-theoretical value. We use therefore a newly proposed variant called Monte-Carlo Tree Search Solver (MCTS-Solver) [18] in our MC-LOA program. The MCTS-Solver is able to prove the game-theoretical value of a position. The backpropagation and selection mechanisms have been modified for this variant.

3.2 The Four Strategic Steps

The four strategic steps of MCTS-Solver are discussed in detail below. We will demonstrate how each of these steps is used in our MC-LOA program.

Selection. Selection picks a child to be searched based on previously gained information. It controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation). On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration).

We use the UCT (**U**pper **C**onfidence Bounds applied to **T**rees) strategy [12], enhanced with Progressive Bias (PB [7]). UCT is easy to implement and used in many Monte-Carlo Go programs. PB is a technique to embed the domain-knowledge bias into the UCT formula. UCT with PB works as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the child k of the node p that satisfies Formula 1:

$$k \in argmax_{i \in I} \left(v_i + \sqrt{\frac{C \times \ln n_p}{n_i}} + \frac{W \times P_{mc}}{n_i + 1} \right) \quad , \tag{1}$$

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which must be tuned experimentally. $\frac{W \times P_{mc}}{n_i+1}$ is the PB part of the formula. W is a constant, which must be set manually (here $W = 10$). P_{mc} is the *transition probability* of a move category mc [15].

For each move category (e.g., capture, blocking) the probability that a move belonging to that category will be played is determined. The probability is called the *transition probability*. This statistic is obtained off-line from game records of matches played by expert players. The transition probability for a move category c is calculated as follows:

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \quad , \tag{2}$$

where $n_{played(mc)}$ is the number of game positions in which a move belonging to category mc was played, and $n_{available(mc)}$ is the number of positions in which moves belonging to category mc were available.

The move categories of our MC-LOA program are similar to the ones used in the Realization-Probability Search of the program MIA [17]. They are applied in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim, and the central 2×2 board. Finally, moves are further classified based on the number of squares traveled away from or towards the center-of-mass.

This selection strategy is applied only at nodes with a visit count higher than a certain threshold T (here 5) [9]. If the node has been visited fewer times than this threshold, the next move is selected according to the *simulation strategy* discussed in the next strategic step.

For all the children of a current leaf node we check whether they lead to a direct win for the player to move. If there is such a move, we stop searching at this node and set the node's value. This check at the leaf node must be performed because otherwise it could take many simulations before the child leading to a mate-in-one is selected and the node is proven. Experiments conducted in the past revealed that this check improved both the playing and solving strength of the engine.

Play-out. The play-out step begins when we enter a position that is not yet a part of the tree. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *simulation strategy*. It is well-known that the use of an adequate simulation strategy improves the level of play significantly [10,11]. The main idea is to play interesting moves according to heuristic knowledge. We describe the simulation strategies in detail in the next section.

Expansion. Expansion is the strategic task that decides whether nodes will be added to the tree. Here, we apply a simple rule: one node is added per simulated game [9]. The added leaf node L corresponds to the first position encountered during the traversal that was not already stored.

Backpropagation. Backpropagation is the procedure that propagates the *result* of a simulated game k back from the leaf node L, through the previously traversed node, all the way up to the root. The result is scored positively ($R_k = +1$) if the game is won, and negatively ($R_k = -1$) if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to the *value* v_L of a node. Here, it is computed by taking the average of the results of all simulated games made through this node [9], i.e., $v_L = (\sum_k R_k)/n_L$.

In addition to backpropagating the values $\{1,0,-1\}$, MCTS-Solver also propagates the game-theoretical values ∞ or $-\infty$. The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax. If the selected move (child) of a node returns ∞, the node is a win. To prove that a node is a win, it suffices to prove that one child of that node is a win. Because of negamax, the value of the node will be set to $-\infty$. In the case that the selected child of a node returns $-\infty$, all its siblings have to be checked. If their values are also $-\infty$, the node is a loss. To prove that a node is a loss, we must prove that all its children lead to a loss. Because of negamax, the node's value will be set to ∞. In the case that one or more siblings of the node have a different value, we cannot prove the loss. Therefore, we will propagate -1, the result for a lost game, instead of $-\infty$, the game-theoretical value of a position. The node will be updated according to the backpropagation strategy as described previously.

3.3 Parallelization

The parallel version of our MC-LOA program uses the so-called "single-run" parallelization [6], also called *root parallelization* [8]. It consists of building multiple MCTS trees in parallel, with one thread per tree. These threads do not share information with each other. When the available time is up, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. Based on this grand total, the best move is selected. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. For a small number of threads, root parallelization performs remarkably well in comparison to other parallelization methods [6,8].

4 Simulation Strategies

In both the selection and the play-out steps move categories together with their associated transition probabilities are used to bias the move selection. In this section we introduce four simulation strategies for further biasing and enhancing the simulation roll-outs. They are *Evaluation Cut-Off*, *Corrective*, *Greedy*, and *Mixed*, and are discussed in detail in Subsect. 4.1 to 4.4, respectively.

4.1 Evaluation Cut-Off

The *Evaluation Cut-Off* strategy stops a simulated game before a terminal state is reached if, according to a heuristic knowledge, the game is judged to be effectively over. In general, once a LOA position becomes quite lopsided, an evaluation function can return a quite trustworthy score, more so than even elaborate simulation strategies. The game can thus be safely terminated both earlier and with a more accurate score than if continuing the simulation (which might, e.g., fail to deliver the win).

We use the MIA 4.5 evaluation function [19] for this purpose. When the evaluation function gives a value that exceeds a certain threshold (e.g., 700 points), the game is scored as a win. Conversely, if the evaluation function gives a value that is below a certain threshold (e.g., -700 points), the game is scored as a loss. For efficiency reasons the evaluation function is called only every 3 plies, starting at the second ply (thus at 2, 5, 8, 11 etc.). This strategy is applied solely in the play-out phase. We remark that a similar strategy was already described by Winands *et al.* in [18]. The Amazons program InvaderMC [13] also terminates simulations early based on an evaluation score. The difference is that in InvaderMC the simulation stops after a fixed length (and subsequently is scored based on the value of the evaluation function), whereas in our approach the simulation may terminate at any time.

4.2 Corrective

One known disadvantage of simulation strategies is that they may draw and play a move which immediately ruins a perfectly healthy position. Embedding domain knowledge, e.g., by the use of Progressive Bias, somewhat alleviates the disadvantage.

In the *Corrective* strategy we use the evaluation function to bias the move selection towards minimizing the risk of choosing an obviously bad move. This is done in the following way. First, we evaluate the position for which we are choosing a move. Next, we generate the moves and scan them to obtain their

```
correctiveStrategy(board){

    defaultValue = evaluate(board);
    moveList = generateMoves();
    scoreSum = 0;

    foreach(Move m in moveList){
        value = evaluate(board, m);
        if (value > bound)
            return m;
        else if (value <= defaultValue)
            m.score = Epsilon;
        else
            m.score = m.getMoveCategoryWeight(board);
        scoreSum += m.score;
    }

    scoreSum = scoreSum*random();
    foreach(Move m in moveList){
        scoreSum -= m.score;
        if(scoreSum <= 0)
            return m;
    }
}
```

Fig. 2. Pseudo code for the Corrective strategy

weights. If the move leads to a successor which has a lower evaluation score than its parent, we set the weight of a move to a preset minimum value (close to zero). If a move leads to a win, it will be immediately played. The pseudo code for this strategy is given in Fig. 2.

4.3 Greedy

In the *Greedy* strategy the evaluation function is more directly applied for selecting moves: the move leading to the position with the highest evaluation score is selected. However, because evaluating every move is time consuming, we evaluate only moves that have a good potential for being the best. For this strategy it means that only the k-best moves according to their transition probabilities are fully evaluated. As in the Evaluation Cut-Off strategy, when a move leads to a position with an evaluation over a preset threshold, the play-out is stopped and scored as a win. Finally, the remaining moves — which are not heuristically evaluated — are checked for a mate. The pseudo code for the Greedy strategy is given in Fig. 3.

4.4 Mixed

A potential weakness of the Greedy strategy is that despite a small random factor in the evaluation function, it is too deterministic. The *Mixed* strategy combines

```
Greedy(Board b){

    moveList = generateMoves();
    assignAndSort(moveList);
    counter = 0;

    foreach(Move m in moveList){
        if(counter < k){
            value = evaluate(board, m);
            if(value > bound){
                return m;
            }
            if(value > max){
                best = m;
                max = value;
            }
        }
        else {
            if(evaluateWin(board, m)) {
                return m;
            }
        }
        counter++;
   }
   return best;
}
```

Fig. 3. Pseudo code for the Greedy strategy

the Corrective strategy and the Greedy strategy. The Corrective strategy is used in the selection step, i.e., at tree nodes where a simulation strategy is needed (i.e., $n < T$), as well as in the first position entered in the play-out step. For the remainder of the play-out the Greedy strategy is applied.

5 Experiments

In this section we evaluate the performance of the four aforementioned simulation strategies by letting them play against themselves and the LOA program MIA. First, we briefly explain MIA in Subsect. 5.1. Next, several settings of the Evaluation Cut-Off strategy are tested in Subsect. 5.2. Subsequently, in order to test the quality of the four simulation strategies, we match them in a round-robin tournament in Subsect. 5.3. Finally, in Subsect. 5.4 we evaluate the playing strength of a MC-based LOA program, using the best settings found in previous subsections, against the $\alpha\beta$-based program MIA.

In the following experiments each match data point represents the result of 1,000 games, with both colors played equally. A standardized set of 100 three-ply starting positions [2] was used, with a small random factor in the evaluation function preventing games from being repeated. The thinking time was set to 1 second per move. All experiments were performed on an AMD Opteron 2.2 GHz computer.

5.1 MIA

MIA is a world-class LOA program, which won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad. It is considered the best LOA-playing entity in the world. All our experiments were performed using the latest and strongest version of the program, MIA 4.5.[1] MIA performs an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced-Realization-Probability-Search framework [17]. The program uses state-of-the-art $\alpha\beta$ enhancements [16].

5.2 Parameter Tuning

In the first series of experiments we tested different cut-off bounds for the Evaluation Cut-Off strategy. For each setting, a program using the Cut-Off strategy played a match against three other programs. The results are given in Table 1. The No-Bound strategy produces moves in the same way as the Evaluation Cut-Off strategy, but always plays simulations out to the end. As we see, such a play-out strategy loses the majority of its games against every setting of the Evaluation Cut-Off strategy: the higher the bound the more the No-Bound strategy loses. Of interest is the relatively good performance of even a bound value of 0, meaning practically that every simulation is cut-off and scored after 2-ply. Nonetheless, under this setting Evaluation Cut-Off still wins most of its games against the No-Bound strategy. The early termination allows more simulations to be performed in the same amount of time, e.g., using a cut-off value of 700 results in over twice as many simulations.

Tuning the bound parameter against a similar (weaker) MC-LOA program may lead to a suboptimal value. Therefore we also played the program against two MIA versions. The first used the MIA III evaluator and the second the MIA 4.5 evaluator. Both versions used the latest search engine. We see in Table 1 that the best bound parameter is somewhere around 600–700. The value of 700 for the bound parameter will be used for the remaining experiments. Moreover, the relative good performance of the 0 setting is again remarkable. What is different from the runs against the No-Bound strategy is that the performance starts to deteriorate when the bound exceeds 1000. Next, the Evaluation Cut-Off strategy significantly improves the way how well simulations do against an $\alpha\beta$ program. For example, whereas the two versions of MIA win 99% of their games against the No-Bound strategy, MIA III and MIA 4.5 only win 22% and 72% of the games against the best settings of the Evaluation Cut-Off strategy. For a proper comparison, the average length and the speed of the play-outs are given in the last two rows of Table 1, respectively.

[1] The program can be found at: http://www.personeel.unimaas.nl/m-winands/loa/

Table 1. 1000-game match results

Bound	0	100	200	400	600	700	800	1000	1200	1400	No-Bound
No-Bound	150.5	149.0	155.5	118.0	112.0	99.0	86.5	57.0	47.0	22.0	X
MIA III	246.0	246.0	227.5	231.5	218.0	253.5	247.5	335.0	398.0	510.5	998
MIA 4.5	794.0	795.0	776.0	768.5	720.0	717.0	754.0	781.0	834.5	868.0	999
Avg. Game Len.	2	2.67	3.56	5.85	8.45	9.83	11.17	13.94	16.65	19.18	53.7
Games per Sec.	10074	9242	8422	6618	5260	4659	4211	3507	2995	2611	2060

5.3 Round-Robin Experiments

In the second series of experiments we quantify the performance of the four simulation strategies in a round-robin tournament. The results are given in Table 2. Surprisingly, the heavily evaluation-function based Greedy strategy was the weakest of the four. The Corrective strategy was better than the Evaluation Cut-Off and Greedy strategy. But, the Mixed strategy, a combination of Corrective and Greedy, outperformed the other ones. The latter result shows that the evaluation function can be directly used for selecting moves as done by Greedy, but not at the start of a simulation. The first moves should be highly randomized.

Table 2. Tournament results

Strategy	Evaluation Cut-Off	Corrective	Greedy	Mixed
Evaluation Cut-Off	-	442.5	598.0	325.5
Corrective	557.5	-	674.0	362.0
Greedy	402.9	326.0	-	167.0
Mixed	674.5	638.0	833.0	-

5.4 MC-LOA vs. MIA

Finally, in the third series of experiments we matched the MC-LOA program using the Mixed strategy against the $\alpha\beta$-based program MIA. The thinking time was set to 5 seconds per move.

The results are given in Table 3. We see from row two that the regular MC-LOA program played almost as well as MIA, receiving a 46% winning score. One nice benefit of MCTS is that it can be parallelized quite easily compared to $\alpha\beta$ search. We tested a two- and four-threaded MC-LOA program against (a single-threaded) MIA and they won 56% and 60%, respectively. We do not have a parallel version of MIA, however, we ran an experiment where the two-threaded MC-LOA program competed against MIA. Here MIA was given 50% more time (simulating a search efficiency increase of 50% if MIA were to be given two processors). A 1,000 game match resulted in a 52% winning percentage for MC-LOA.

It is beyond the scope of paper to investigate to what extent MCTS scales better than $\alpha\beta$. To give an indication, experiments revealed that for 1 second per move MC-LOA won 42% of the games, whereas for 5 seconds per move MC-LOA already won 46% of the games.

Table 3. 1,000-game match results

	Score	Win %	Winning ratio
1 × MC-LOA vs. MIA 4.5	458.0 - 542.0	46%	0.85
2 × MC-LOA vs. MIA 4.5	563.5 - 436.5	56%	1.29
4 × MC-LOA vs. MIA 4.5	602.5 - 397.5	60%	1.52

6 Conclusion and Future Research

In this paper we investigated how to use a positional evaluation function to enhance a simulation-based LOA program. Four different simulation strategies were designed, called Evaluation-Cut Off, Corrective, Greedy, and Mixed.

Our experimental results showed that the Mixed strategy of playing greedily in the play-out phase, but exploring more in the earlier selection phase, although in a way such that it avoids moves that immediately deteriorate the position, works the best. Experiments also showed that applying an evaluation function to stop simulations when a game is judged to be effectively over, resulted in a significant increase in both the number of simulations and playing strength.

Collectively, these enhancements resulted in our simulation-based MC-LOA program to play at a comparable level as the world-class $\alpha\beta$-based program MIA. Moreover, equipped with a simple root parallelization the MC-LOA program outperformed MIA both when using two and four threads. Based on the greatly improved playing strength that we witnessed in the MC-LOA program when adding the enhancements proposed above, we believe that it is only a matter of time until simulation-based programs will significantly outperform $\alpha\beta$-based programs in the game LOA. This is an important milestone for MCTS, because up until now the traditional game-tree search approach has generally been considered to be better suited for the game LOA.

As a future research we plan to continue work on enhancing the simulation strategies both by tuning the various parameters involved and by combining the strategies in more elaborate ways.

References

1. Abramson, B.: Expected-outcome: A general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence 12(2), 182–193 (1990)
2. Billings, D., Björnsson, Y.: Search and knowledge in Lines of Action. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 231–248. Kluwer Academic Publishers, Boston (2003)
3. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 159–174. Kluwer Academic Publishers, Boston (2003)
4. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)

5. Cazenave, T., Borsboom, J.: Golois Wins Phantom Go Tournament. ICGA Journal 30(3), 165–166 (2007)
6. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computer Games Workshop 2007 (CGW 2007), Universiteit Maastricht, Maastricht, The Netherlands, pp. 93–101 (2007)
7. Chaslot, G.M.J.-B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
8. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J.: Parallel monte-carlo tree search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
9. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
10. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Fox, D., Gomes, C.P. (eds.) Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, pp. 259–264 (2008)
11. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference on Machine Learning (ICML), pp. 273–280. ACM, New York (2007)
12. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
13. Lorentz, R.J.: Amazons discover monte-carlo. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 13–24. Springer, Heidelberg (2008)
14. Sackson, S.: A Gamut of Games. Random House, New York (1969)
15. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 132–144 (2002)
16. Winands, M.H.M.: Informed Search in Complex Games. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands (2004)
17. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)
18. Winands, M.H.M., Björnsson, Y., Saito, J.-T.: Monte-carlo tree search solver. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 25–36. Springer, Heidelberg (2008)
19. Winands, M.H.M., van den Herik, H.J.: MIA: a world champion LOA program. In: The 11th Game Programming Workshop in Japan (GPW 2006), pp. 84–91 (2006)
20. Zhang, P., Chen, K.-H.: Monte Carlo Go capturing tactic search. New Mathematics and Natural Computation 4(3), 359–367 (2008)

Monte-Carlo Kakuro

Tristan Cazenave

LAMSADE
Université Paris-Dauphine
Paris, France
cazenave@lamsade.dauphine.fr

Abstract. Kakuro consists in filling a grid with integers that sum up to predefined values. Sums are predefined for each row and column and all integers have to be different in the same row or column. Kakuro can be modeled as a constraint satisfaction problem. Monte-Carlo methods can improve on traditional search methods for Kakuro. We show that a Nested Monte-Carlo Search at level 2 gives good results. This is the first time a nested search of level 2 gives good results for a Constraint Satisfaction problem.

1 Introduction

Kakuro, also known as Cross Sums is a popular NP-complete puzzle [5]. It consists of a predefined grid containing open cells. the left edge and the upper edge of the grid contain a column and a row of cells with a predefined number. Each open cell has to be filled with an integer between 1 and 9. All cells in the same row and all cells in the same column have to contain different integers. The sum of the integers of a row has to match the predefined number, as well as the sum of the integers of a column. Table 1 gives an example of a 5x5 Kakuro problem. The sum of the integers of the first row has to be 18, the sum of the integers of the last column has to be 24. Moreover, it is possible that a predefined number of holes is given in advance. Section 2 details the search algorithms used for Kakuro. In Section 3 we present experimental results. Section 4 contains our conclusion.

Table 2 gives a solution to the problem of table 1.

Table 1. A 5x5 Kakuro puzzle

	24	25	20	26	24
18					
26					
28					
26					
21					

Table 2. A solution to the previous Kakuro puzzle

	24	25	20	26	24
18	1	7	5	3	2
26	4	5	3	8	6
28	5	6	7	2	8
26	8	4	1	6	7
21	6	3	4	7	1

2 Search Algorithms

The search algorithms we have tested are (1) Forward Checking, (2) Iterative Sampling, (3) Meta Monte-Carlo search, and (4) Nested Monte-Carlo Search. They are presented below. We complete the section by some remarks on choosing a variable (2.5) and choosing a value (2.6).

2.1 Forward Checking

The Forward Checking algorithm consists in reducing the set of possible values of the free variables after each assignment of a value to a given variable. It reduces the domains of the free variables that appear in the same constraints as the assigned variable.

In Kakuro, each time a value is assigned to a variable, the value is removed from the domain of the free variables that are either in the same row or in the same column as the assigned variable.

Moreover, the sum S_{row} of all the assigned variables in a row is computed, then the maximum possible value $Maxval$ for any variable in the row is computed by subtracting S_{row} from the goal sum of the row. All values that are greater than $Maxval$ are removed from the domains of the free variables of the row. If all the variables of the row have been assigned the sum is compared to the target sum and if it is different, the assignment is declared inconsistent.

A similar domain reduction and consistency check is performed for the free variables in the column of the assigned variable.

Much more elaborate consistency checks could be performed and would improve all the algorithms presented in this paper. However, our point in this paper is not about elaborate consistency checks but rather about the interest of nested search.

A Forward Checking search is a depth-first search that chooses a variable at each node, tries all the values in the domain of this variable and recursively calls itself until a domain is empty or a solution is found.

The pseudo code for Forward Checking is as follows:

```
bool ForwardChecking ()
  if no free variable then
    return true
  choose a free variable var
  for all values in the domain of var
```

```
    assign value to var
    update the domains of the free variables
    if no domain is empty or inconsistent then
      if (ForwardChecking ()) then
        return true
  return false
```

2.2 Iterative Sampling

Iterative Sampling uses Forward Checking to update the possible values of the free variables. A sample consists in choosing a variable, assigning a possible value to it, updating the domains of the other free variables, and looping until a solution is found or a variable with an empty domain is found. Iterative sampling performs samples until a solution is found or the allocated time for the search is elapsed.

The sample function that we give returns the number of free variables that are left when a variable has an empty domain because this value is used as the score of a sample by other algorithms.

The pseudo code for sampling is as follows:

```
int sample ()
  while true
    choose a free variable var
    choose a value in the domain of var
    assign value to var
    update the domains of the free variables
    if a domain is empty or inconsistent then
      return 1 + number of free variables
    if no free variable then
      return 0
```

The Iterative Sampling algorithm consists in repeatedly calling sample.

```
bool iterativeSampling ()
  while time left
    if sample () equals 0 then
      return true
  return false
```

2.3 Meta Monte-Carlo Search

Rollouts were successfully used by Tesauro and Galperin to improve their Backgammon program [6]. They consist in playing games according to an algorithm that chooses the moves to play. The scores of the games are then used to choose a move instead of directly using the base algorithm. A related algorithm that has multiple levels is Reflexive Monte-Carlo search [2] which has been used to find long sequences at Morpion Solitaire. Reflexive Monte-Carlo search consists in (1) playing random playouts at the base level, and then (2) playing a few games at the lower level of a search in order to find the best move at the current level of the search. At Morpion Solitaire, games at the meta level give better results than games at the lower level.

A Meta Monte-Carlo Search (1) tries all possible assignments of the variable, (2) plays a sample after each assignment, and (3) chooses the value that has the best sample score. The algorithm memorizes the best sample so as to follow it in subsequent moves if no better sample has been found.

The pseudo code of Meta Monte-Carlo Search is as follows:

```
1  int metaMonteCarlo ()
2    best score = number of free variables
3    while true
4      choose a free variable var
5      for all values in the domain of var
6        assign value to var
7        update the domains of the free variables
8        if a domain is empty or inconsistent then
9          score = 1 + number of free variables
10       else
11         score = sample ()
12       if score < best score then
13         best score = score
14         best sequence = {{var,value},sample sequence}
15     var = pop variable of the best sequence
16     value = pop value of the best sequence
17     assign value to var
18     update the domains of the free variables
19     if a domain is empty or inconsistent then
20       return 1 + number of free variables
21     if no free variable or best score equals 0 then
22       return 0
```

At line 14, when a sample has found a new best sequence, it is memorized. A sequence consists of an ordered list of variables and values that have been chosen during the sample. It is analogous to a sequence of moves in a game.

The algorithm can be used with any maximum allocated time, repeatedly calling it until a solution is found or the time is elapsed, as for Iterative Sampling.

```
bool iterativeMetaMonteCarlo ()
  while time left
    if metaMonteCarlo () equals 0 then
      return true
  return false
```

2.4 Nested Monte-Carlo Search

Nested Monte-Carlo Search [3] pushes further the meta Monte-Carlo approach, using multiple meta-levels of nested Monte-Carlo searches. This approach is similar to previous approaches that attempt to improve an heuristic of a solitaire card game with nested calls [7,1]. These algorithms use a base heuristic which is improved with nested calls, whereas Nested Monte-Carlo Search uses random moves at the base level instead. Nested Monte-Carlo Search is an algorithm that uses no domain specific knowledge and which is widely applicable. However adding domain specific knowledge will probably improve it. For example, at Kakuro pruning more values using stronger consistency checks would certainly improve both the Forward Checking algorithm and the Nested Monte-Carlo search algorithm.

The application of Nested Monte-Carlo Search to Constraint Satisfaction is as follows.

```
1  int nested (level)
2    best score = number of free variables
3    while true
4      choose a free variable var
5      for all values in the domain of var
6        assign value to var
7        update the domains of the free variables
8        if a domain is empty or inconsistent then
9          score = 1 + number of free variables
10       else if level is 1
11         score = sample ()
12       else
13         score = nested (level - 1)
14       if score < best score then
15         best score = score
16         best sequence = {{var,value},level-1 sequence}
17     var = pop variable of the best sequence
18     value = pop value of the best sequence
19     assign value to var
20     update the domains of the free variables
21     if a domain is empty  or inconsistent then
22       return 1 + number of free variables
23     if no free variable or best score equals 0 then
24       return 0
```

Meta Monte-Carlo search is a special case of Nested Monte-Carlo Search at level 1.

It is important to memorize the best sequence of moves in Nested Monte-Carlo Search. This is done at line 16 of the nested function. At that line, if the search of the underlying level has found a new best sequence, the sequence is copied as the best sequence of the current level.

Nested Monte-Carlo Search parallelizes very well, for the Morpion Solitaire disjoint version, speedups of 56 for 64 processors were obtained [4].

The algorithm can be used with any allocated time, repeatedly calling it at a given level.

2.5 Choosing a Variable

When choosing a variable, the usual principle is to choose the variable that will enable to find that there is no solution under the node as fast as possible. A common, general and efficient heuristic is to choose the variable that has the smallest domain size. This is the heuristic we have used for all the algorithms presented in this paper.

2.6 Choosing a Value

When choosing a value, the usual principle is to choose the value that has the most chances of finding a solution because if there is no solution, all values have to be tried in order to prove that there is no solution. In this paper we choose values at random among possible values of a variable. However, Nested Monte-Carlo Search mainly consists in obtaining much information about the interestingness of all values before choosing one, so it can also be considered as an algorithm that carefully selects values to be tried.

3 Experimental Results

In this section we explain how problems have been generated. We then give the results of running the algorithm on various problems. We also evaluate the influence of the number of possible values on problem hardness.

3.1 Problem Generation

In order to generate a problem, a search is used to generate a complete grid of a given size with the constraint that all values are different in the same column or row. Then the sum of each row and each column is computed. Then values are randomly removed until the desired percentage of holes is reached. For each percentage of holes, 100 problems have been generated.

3.2 Comparison of Algorithms

The four algorithms that were tested are Forward Checking, Iterative Sampling, Nested Monte-Carlo Search at level 1 and level 2.

In order to estimate the problem difficulties, these four algorithms were tested on all percentages of holes of 10x10 grids with values ranging from 1 to 11. Figure 1 gives the number of problems solved for each percentage and each algorithm using a timeout of 10 seconds per problem. Figure 2 gives the total time used by each algorithm for the same problems. From these figures it is clear that (1) Nested Monte-Carlo search at level 2 easily solves almost all the problems in less than 10 seconds, and (2) Forward Checking and Iterative Sampling solve almost no problem within 10 seconds when the problems have more than 80% of free variables.

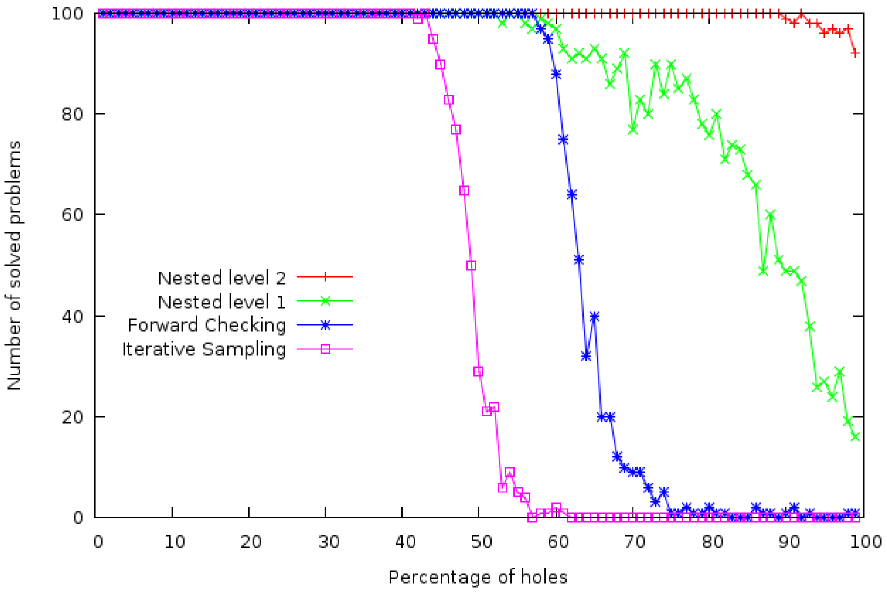


Fig. 1. Number of solved problems for different percentage of holes and different algorithms with a timeout of 10 seconds per problem, 10x10 grids, values ranging from 1 to 11

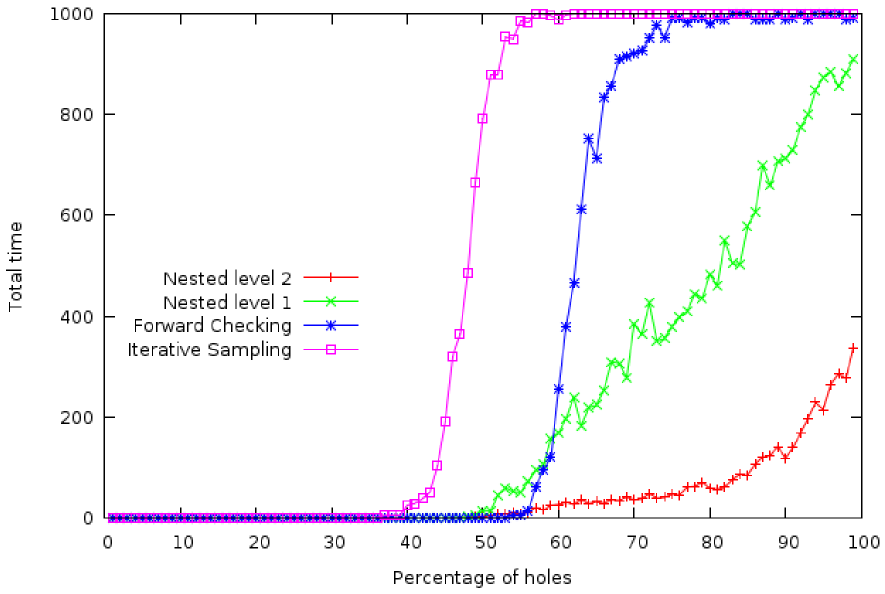


Fig. 2. Time spent for all problems for different percentage of holes and different algorithms with a timeout of 10 seconds per problem, 10x10 grids, values ranging from 1 to 11

Table 3. Comparison of different algorithms for empty 6x6 grids, values ranging from 1 to 7, timeout of 1,000 seconds

Algorithm	Problems solved	Total time
Nested Level 2	100 out of 100	0.91 s.
Nested Level 1	100 out of 100	1.42 s.
Iterative Sampling	100 out of 100	424.21 s.
Forward Checking	98 out of 100	9,433.98 s.

Table 4. Comparison of different algorithms for empty 8x8 grids, values ranging from 1 to 9, timeout of 1,000 seconds

Algorithm	Problems solved	Total time
Nested Level 2	100 out of 100	17.85 s.
Nested Level 1	100 out of 100	78.30 s.
Iterative Sampling	10 out of 100	94,605.16 s.
Forward Checking	8 out of 100	92,131.18 s.

We can see that the problem difficulty increases with the number of holes, the most difficult problems being the empty grids problems. This is different from a closely related problem, the quasi group completion problem. This problem also consists in filling a grid with different values on each row and each column, but there is no constraints on the sum of the values and there are as many values as the size of the column or the row. The quasi group completion problem is easy for low and high percentages of holes and hard for intermediate percentages. In our experiments, Iterative Sampling very easily solves all quasi group completion problems, with any percentage of holes, up to size 10x10. Kakuro is harder to solve than quasi group completion for the same problem size, and the problems hardness does not have the same repartition.

Moreover Nested Monte-Carlo Search at level 2 is better than Nested Monte-Carlo Search at level 1 which is better than Forward Checking which is in turn better than Iterative Sampling.

We now compare algorithms giving them more time (1,000 seconds per problem) on empty 6x6 grids (36 free variables) since empty grids are the most difficult problems. Possible values range from 1 to 7. Table 3 gives the number of problems solved and the time to solve them for the different algorithms. We see that Nested Monte-Carlo Search is still the best algorithm, however Iterative Sampling becomes much better than Forward Checking on 6x6 empty grids.

In order to test more difficult problems we repeated the experiment for empty 8x8 grids with values ranging from 1 to 9. The results are given in table 4. Nested Monte-Carlo Search at level 2 is still the best algorithm while Iterative Sampling and Forward Checking perform badly. We can observe that most of the time of the Forward Checking algorithm is spent on problems that are not solved (92 problems for 92,000 seconds) while Iterative Sampling solves two more problems but takes more time.

Table 5. Solving empty 8x8 grids with different numbers of possible values, timeout of 1,000 seconds

Algorithm	Problems solved	Possible Values	Total time
Nested Level 2	100 out of 100	[1,9]	17.85 s.
Nested Level 2	100 out of 100	[1,10]	1,939.51 s.
Nested Level 2	100 out of 100	[1,11]	1,166.13 s.
Nested Level 2	100 out of 100	[1,12]	1,214.29 s.
Nested Level 2	99 out of 100	[1,13]	1,688.51 s.
Nested Level 2	100 out of 100	[1,14]	629.47 s.
Nested Level 2	100 out of 100	[1,15]	672.90 s.
Nested Level 2	100 out of 100	[1,16]	603.38 s.
Nested Level 2	100 out of 100	[1,17]	429.26 s.
Nested Level 2	100 out of 100	[1,18]	579.99 s.

3.3 Influence of the Number of Values

The next experiment consists in estimating the evolution of the difficulty of Kakuro problems with the number of possible values. We solved 100 empty 8x8 grids with Nested Monte-Carlo Search level 2 for 9 to 18 possible values. The results are given in table 5. We see that the difficulty starts to increase with the number of possible values and then decreases when enough values are possible.

4 Conclusion

We have compared Forward Checking, Iterative Sampling, and Nested Monte-Carlo Search on Kakuro problems. Nested Monte-Carlo search at level 2 gives the best results. We have also shown the difficulty of Kakuro problems given their number of holes and number of possible values.

Future work include testing the algorithms with stronger consistency checks and comparing them on other problems.

Acknowledgments

This work has been supported by French National Research Agency (ANR) through COSINUS program (project EXPLO-RA ANR-08-COSI-004).

References

1. Bjarnason, R., Tadepalli, P., Fern, A.: Searching solitaire in real time. ICGA Journal 30(3), 131–142 (2007)
2. Cazenave, T.: Reflexive Monte-Carlo search. In: Computer Games Workshop, Amsterdam, The Netherlands, pp. 165–173 (2007)

3. Cazenave, T.: Nested Monte-Carlo search. In: IJCAI 2009, Pasadena, USA, pp. 456–461 (2009)
4. Cazenave, T., Jouandeau, N.: Parallel nested Monte-Carlo search. In: NIDISC Workshop, Rome, Italy (2009)
5. Kendall, G., Parkes, A., Spoerer, K.: A survey of NP-complete puzzles. ICGA Journal 31(1), 13–34 (2008)
6. Tesauro, G., Galperin, G.: On-line policy improvement using monte-carlo search. In: Advances in Neural Information Processing Systems, vol. 9, pp. 1068–1074. MIT Press, Cambridge (1996)
7. Yan, X., Diaconis, P., Rusmevichientong, P., Roy, B.V.: Solitaire: Man versus machine. In: Advances in Neural Information Processing Systems, vol. 17, pp. 1553–1560. MIT Press, Cambridge (2005)

A Study of UCT and Its Enhancements in an Artificial Game

David Tom and Martin Müller

Department of Computing Science, University of Alberta, Edmonton, Canada, T6G 2E8
{dtom,mmueller}@cs.ualberta.ca

Abstract. Monte-Carlo tree search, especially the UCT algorithm and its enhancements, have become extremely popular. Because of the importance of this family of algorithms, a deeper understanding of when and how the different enhancements work is desirable. To avoid the hard to analyze intricacies of tournament-level programs in complex games, this work focuses on a simple abstract game, which is designed to be ideal for history-based heuristics such as RAVE. Experiments show the influence of game complexity and of enhancements on the performance of Monte-Carlo Tree Search.

1 Introduction

Monte Carlo Tree Search (MCTS), especially in form of the UCT algorithm [1], has become an immensely popular approach for game-playing programs [2]. MCTS has been especially successful in environments for which a good evaluation function is hard to build, such as Go [3] and General Game-Playing [4]. MCTS-based programs are also on par with the best traditional programs in Hex and Amazons [5].

Part of the success of MCTS is due to its enhancements. Methods inspired by Schaeffer's history heuristic [6] include *All-Moves-As-First* (AMAF) [7] and Rapid Action Value Estimation (RAVE) [8]. Whereas the value of a move is typically based on simulations where the move is the first one played, these heuristics use all simulations where the move is played at any point in the game; this produces a low variance estimate that is fast to learn [8]. Methods such as progressive pruning [9] focus MCTS on stronger-looking candidate branches.

While the game-independent algorithms above can be used with minor variations across different games, typical tournament-level programs contain a large number of game-specific enhancements as well, such as opening books and specialized playout policies. Further examples are patterns [3,10] and tactical subgoal solvers in Go, and virtual connections in Hex.

While practical applications abound, up to this point there has been relatively little detailed analysis of the core MCTS algorithm and its enhancements. Gaining a deeper understanding of their behaviour and performance is difficult in the context of complicated programs for complex games. Rigorous testing, evaluation and interpretation of the results is necessary but difficult to do in such environments. A simpler, well-controlled environment seems necessary.

1.1 Research Questions

Since MCTS is a relatively new approach, there is a large number of open research questions, both in theory and in practice. For example,

- How does the performance of an algorithm vary with the complexity and type of game that is played?
- What are the conditions on a game under which a specific enhancement works? How much does it improve MCTS in the best case?
- How should a general framework for Monte-Carlo Tree Search be designed, and how can it then be adapted to a specific game?

Some of these questions are addressed in practice by the FUEGO system [11], an open-source library for games which includes the MCTS engine used for the experiments in this paper. One way to study questions about MCTS in more precision than is possible for real games is to use highly simplified, abstract games for which a complete mathematical analysis is available. Ideally, such games should allow deeper study of the core algorithms while avoiding layers of game-specific complexity in the analysis.

In this paper, a simple artificial game, called Sum of Switches (SOS), is used for an experimental study of MCTS algorithms, in particular, as a close to ideal scenario for the RAVE heuristic. Section 2 introduces and motivates the SOS game model, and discusses related work on analysis of MCTS. Section 3 briefly summarizes relevant parts of the FUEGO framework used in the experiments. Section 4 describes our experiments. Sections 5 and 6 conclude with a discussion of our results and ideas for future work.

2 The Sum of Switches Game

Sum of Switches (SOS) is a number picking game played by two players. The game has one parameter n. In SOS(n) players alternate turns picking one of n possible moves. Each move can only be picked once. The moves have values $\{0, \ldots, n-1\}$, but the values are hidden from the players. The only feedback for the players is whether they win or lose the overall game. After n moves, the game is over. Let s_1 be the sum of all first player's picks, $s_1 = p_{1,1} + \ldots + p_{1,n/2}$, and s_2 the sum of second player's picks, $s_2 = p_{2,1} + \ldots + p_{2,n/2}$. Scoring is similar to the game of Go. The *komi* k is set to the perfect play outcome, $k = (n-1) - (n-2) + \ldots = \lfloor n/2 \rfloor$. The first player wins iff $s_1 - s_2 \geq k$.

The optimal strategy for both players would be to simply choose the largest remaining number at each step. However, since both the move values and the final scoring system are unknown to the players, good moves must be discovered through exploration, by repeated play of the same game.

SOS can be viewed as a generalized multi-armed bandit game. In classical multi-arm bandit problems, each game consists of picking a single arm i out of n possible arms, which leads to an immediate reward X_i, a random variable. The player uses exploration to find the arm with best expected reward, and exploits that arm by playing it. In SOS, one episode consists of playing *all* arms once. The reward X_i for choosing arm i is constant, but is not directly shown to the player. Only the success of all choices relative to the opponents choices is revealed at the end of the episode.

2.1 Related Work and Motivation for SOS

The original UCT paper [1] contains an experiment showing the performance of UCT on the artificial P-game tree model [12]. Each edge representing a move is associated with a random number from a specified range. The value of a leaf node is the sum of the edge values along the path from the root. The value of edges corresponding to opponent moves is negated.

In the SOS model, the value of a move is independent of when and by which player it is chosen. This should represent a best-case scenario for history-based heuristics such as RAVE.

The RAVE heuristic is a frequently used enhancement for MCTS. In contrast to basic Monte-Carlo tree search, it collects statistics over *all* moves played in a simulation. In a game such as SOS, that extra information should be of high quality since moves have the same value independent of when they are played.

In the original work on RAVE [8], Gelly and Silver analyze its performance in the context of computer Go. The weights for the RAVE heuristic were chosen empirically to work well in Go. Empirically, RAVE is shown to have very strong overall performance in Go. However, it causes occasional blunders by introducing a strong bias against the correct move. For example, if a move is very good right now, but very bad if played at any time later in a simulation, RAVE updates would be misleading. Such misleading biases do not exist in the case of SOS.

3 The FUEGO Framework and Its MCTS Implementation

The experiments with SOS use the FUEGO framework [11], which includes the computer Go program with the same name. One component of the FUEGO framework is the game-independent SmartGame library: a set of tools to handle game play, file storage, and game tree search as well as other utility functions. The SmartGame library includes a generic MCTS engine with support for UCT, RAVE, and using prior knowledge. The UCT and RAVE engines are used in the SOS experiments with no modifications. No experiments on utilizing prior knowledge are presented in this paper.

The UCT engine uses the basic UCT formula, with user-defined parameters controlling the UCT behaviour. The parameter c is defined by the user to determine the influence of the UCB bound value; this parameter is usually optimized by hand, but for the purposes of SOS, we chose to keep it at the default value of 0.7.

When RAVE is active, the value of the estimate for a move is determined by a linear combination of the mean value and RAVE value of the move. The weighting function used here is a little different from the one originally proposed in [8], but has been found to work as well as the original formula in FUEGO. The unnormalized weighting of the RAVE estimator is determined by the formula:

$$W_j = \frac{\beta_j w_f w_i}{w_f + w_i \beta_j}$$

the RaveCount β_j represents the number of rave updates of move j. w_i and w_f stand for RaveWeightInitial and RaveWeightFinal; these parameters determine the influence of RAVE relative to the mean value. They are manually set by the user. w_i describes the initial slope of the weighting function and w_f describes its asymptotic bound. As the number of simulations increases, the weight of the RAVE value diminishes relative to the mean value. This formula is designed to lower the mean squared error of the weighted sum; it is optimal when the weight of each estimator is proportional to the inverse of its mean squared error. In practice, the values of RaveWeightInitial is usually kept at the default value of 1.0, and a suitable RaveWeightFinal is found experimentally. RaveWeightInitial is kept at 1.0 as we do not make any assumptions about the accuracy of early RAVE and UCT estimates. The weight W_j is used in the UCT formula in the following manner [11]:

$$MoveValue(j) = \frac{T_j(\alpha)}{T_j(\alpha) + W_j}\bar{X}_j + \frac{W_j}{T_j(\alpha) + W_j}\bar{Y}_j + c\sqrt{\frac{\log \alpha}{T_j(\alpha) + 1}}$$

α represents the number of times the parent node was visited. $\bar{X}_j$ denotes the average reward and $\bar{Y}_j$ the RAVE value of move j. The c term is a constant bias term set by the user and $T_j(\alpha)$ is the MoveCount, the number of times move j has been played at the parent node. Adding 1 to $T_j(\alpha)$ in the bias term avoids a division by 0 in case move j has a RAVE value but $T_j(\alpha) = 0$. In MCTS, the game tree is grown incrementally. In the SmartGame library unexpanded nodes are assigned a *FirstPlayUrgency* value. Large values cause the program to prioritize exploration whereas small values encourage exploitation. The default value of 10000 is used in the experiments, which gives high priority to unexpanded nodes [1].

4 Experiments

The experiments investigate the properties of MCTS with UCT and RAVE. Results are shown for (1) varying the size of the game and the number of simulations used in the search, (2) the influence of RAVE, (3) training on optimal play vs. "good" play, and (4) the effect of misleading RAVE updates. This paper reports our findings thus far, and will hopefully lead to further experiments with MCTS enhancements and improved algorithms. The experiments were performed on 2 GHz i686 computers with 1GB of memory running Linux 2.6.25.14-108.fc9.i686 Fedora release 9 (Sulphur). The FUEGO version used in these experiments was FUEGO release 0.2.

4.1 Game Size and Simulation Limits

The complexity of the SOS game is determined solely by its size. SOS(n) produces a game tree of size $n!$ since transpositions and tree pruning are not present in this model. For example, the complete SOS(10) game tree contains 3628800 leaf nodes. The larger the game is, the more difficult it is for a game-playing program to solve. The performance of the game-playing program is mainly determined by a single parameter s: the number of simulations it is allowed to perform before playing a move. To establish a baseline for the performance of UCT in SOS, experiments varying n and s were performed.

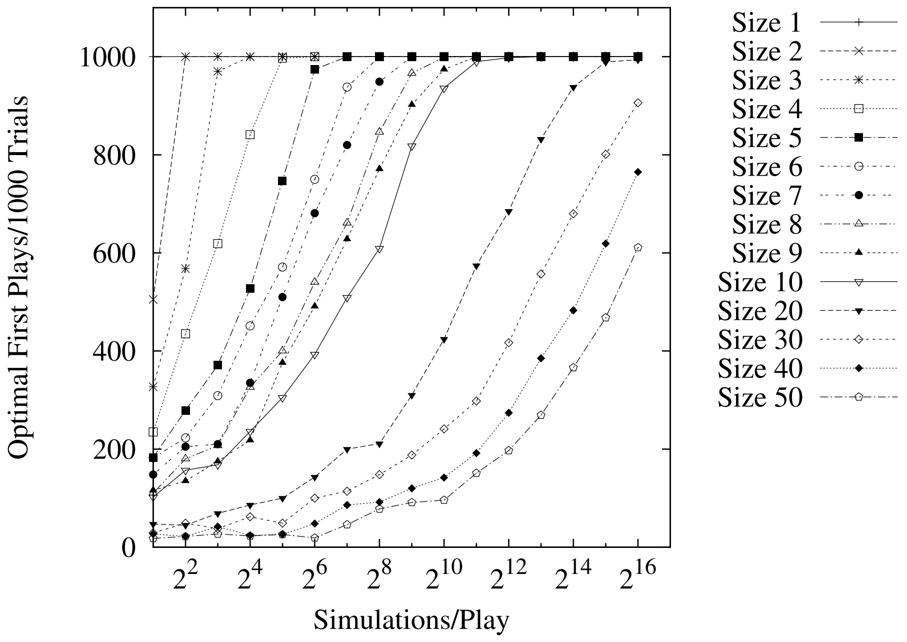


Fig. 1. Plain UCT without enhancements in SOS(n)

Each data point in Figure 1 represents how often the optimal first move was chosen in 1000 trials. For $n < 10$ the program quickly converges to optimal play. In the range $10 \leq n \leq 50$, convergence becomes progressively slower. The convergence rates seem similar to those in [1] for games with a comparable number of leaf nodes. For further experiments, SOS(10) was chosen as a compromise between game difficulty and runtime until convergence.

4.2 RAVE

Figure 2 shows experiments with RAVE. Even low values of RaveWeightFinal such as 16 give noticeable improvements. Large values of RaveWeightFinal show diminishing returns, with 512 producing similar results to 32768 or higher values.

As stated previously, this game is designed as a kind of best-case for RAVE: the relative value between moves is consistent at all stages of the game. In fact, in SOS it is possible and beneficial to base the UCT search exclusively on the RAVE value and ignore the mean value. Figure 2 includes this RAVE-only data as well. Of course, this method would not work in other games where the value of a move depends on the timing when it is played.

4.3 Score Bonus

Score Bonus is an enhancement that differentiates between strong and weak wins and losses. If game results are simply recorded as a 0 or 1, the program does not receive

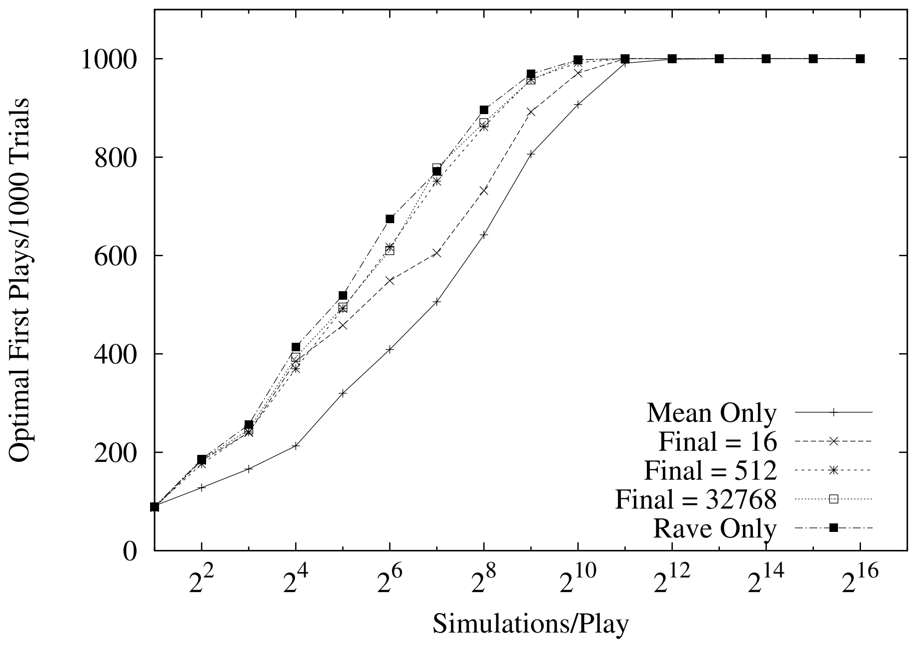


Fig. 2. UCT+RAVE, varying RaveWeightFinal in SOS(10)

any feedback on how close it was to winning or losing. With score bonus, a high win that probably contained many high-scoring moves gets a slightly better evaluation than a close win.

In SOS with score bonus, losses are evaluated in a range from 0 to γ and wins from $1-\gamma$ to 1, for a parameter γ. A minimal win is awarded $1-\gamma$, and a maximal possible win a score of 1. All other game outcomes are scaled linearly in this interval. The values assigned for losses are analogous.

Results for $\gamma = 0.1$, $\gamma = 0.05$, and $\gamma = 0.02$ are shown in Figure 3. Score bonus fails to improve gameplay in SOS. However, it is used in the FUEGO Go program. Unpublished large-scale experiments by Markus Enzenberger showed that small positive values of γ improve the playing strength slightly but significantly for 9×9. Best results were achieved for $\gamma = 0.02$.

4.4 False Updates

While RAVE works very well in SOS and Go, it is not reliable in all games. Since RAVE updates the value of all moves in a winning sequence and ignores temporal information, it can lead the search astray. In situations where specific moves are only helpful at a given time, RAVE can weaken game-play instead of improving it. Suppose that in a game, a certain last move will always lead to a win, but is useless at all other times. The high RAVE value that this move is likely to earn early in simulations is likely

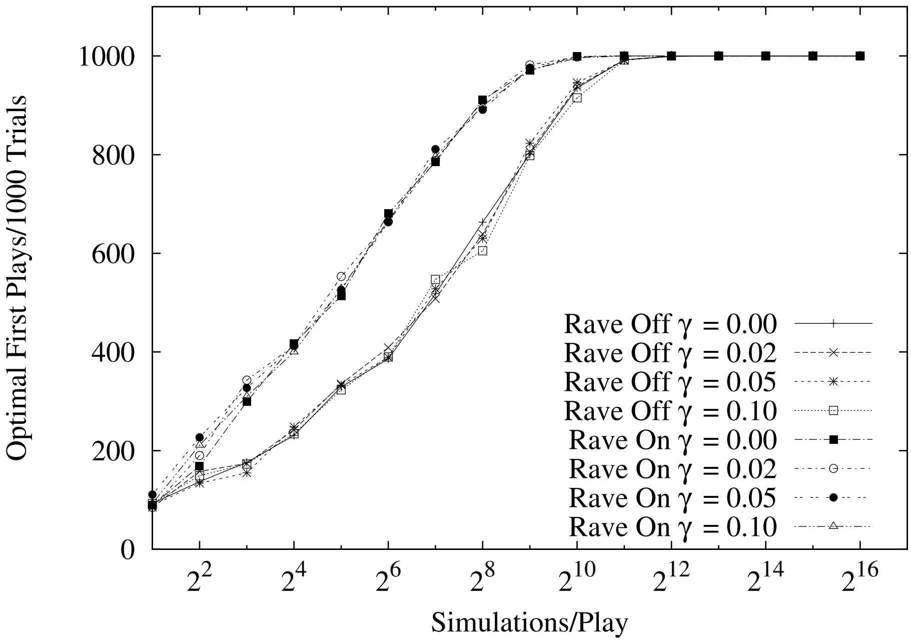


Fig. 3. Graph of Score Bonus results on SOS(10). The RAVE experiments were performed with the RAVE-only settings.

to cause the game-playing program to waste a lot of time exploring paths related to this winning move at higher points in the tree. It is potentially possible for such a situation to result in very poor value estimates when the simulation limit is reached and thus poor play will result.

Experiments involving random false updates can simulate the effect of misleading RAVE values. With a probability of μ, the Rave update for all moves in the current simulation uses the inverse evaluation $InverseEval = 1 - Eval$. $RaveWeightFinal$ was set to a high value in this set of experiments so as to pronounce the effect of the experiment; additionally, this setup also reflects scenarios where little is known about the game, but RAVE is expected to be a strong estimator. The results of these experiments are summarized in Figure 4.

Even with the influence of the mean value as a steadying force, the performance of a program with RAVE influence deteriorates as the value of μ increases. The decay is gradual until μ is about 0.5, where performance drops significantly. RAVE still outperforms plain UCT when the false update rate is between 0 and 0.3. Up to an error rate of 0.5, the error can be interpreted as noise that slows down convergence; error rates above 0.5 have an antagonistic effect upon the RAVE heuristic. Even with $\mu = 0.6$ the performance still improves with the number of simulations. These results suggest that with unbiased noise as provided by false updates above, RAVE is a robust heuristic that is resilient against a reasonable level of error.

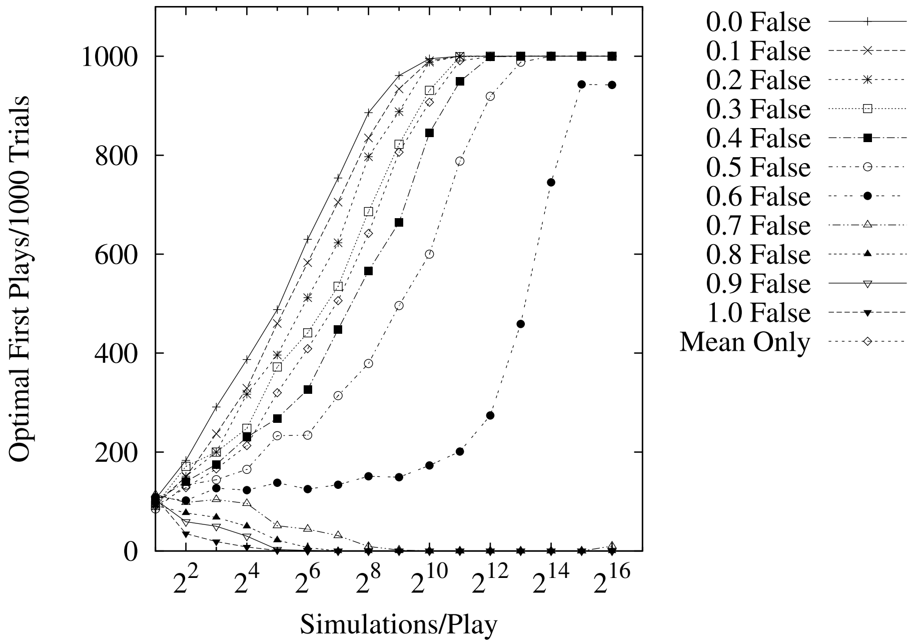


Fig. 4. Effect of False Updates on RAVE with $\mathit{RaveWeightFinal} = 32768$. Experiments performed in SOS(10).

It would be interesting to study a biased version of false updates, that selectively distorts the updates related to specific moves. This may be a model that is closer to what is seen in Go, and present more problems for the search.

Since RAVE-only works well in SOS, it is interesting to see the effect of false updates here. The results in Figure 5 show a similar trend while the error rate is low. The $\mu = 0$ data corresponds to Figure 2, where RAVE-only is better than UCT+RAVE. However, at $\mu = 0.2$ RAVE-only is already slightly worse, and at $\mu = 0.4$, RAVE-only is far worse than the UCT+RAVE version shown in Figure 4. At $\mu = 0.5$ the algorithm behaviour becomes random.

5 Analysis

The experiments studied UCT and two common enhancements, RAVE and Score Bonus. Score Bonus did not produce favourable results in SOS, but had a positive effect in Go. This discrepancy needs further study.

The RAVE experiments show significantly better performance than plain UCT, even with distorted RAVE updates. The experiments suggest that the RAVE heuristic is robust against unbiased noise and performs well even with a fair level of error. However, the RAVE experiments also suggest that performance can be significantly improved

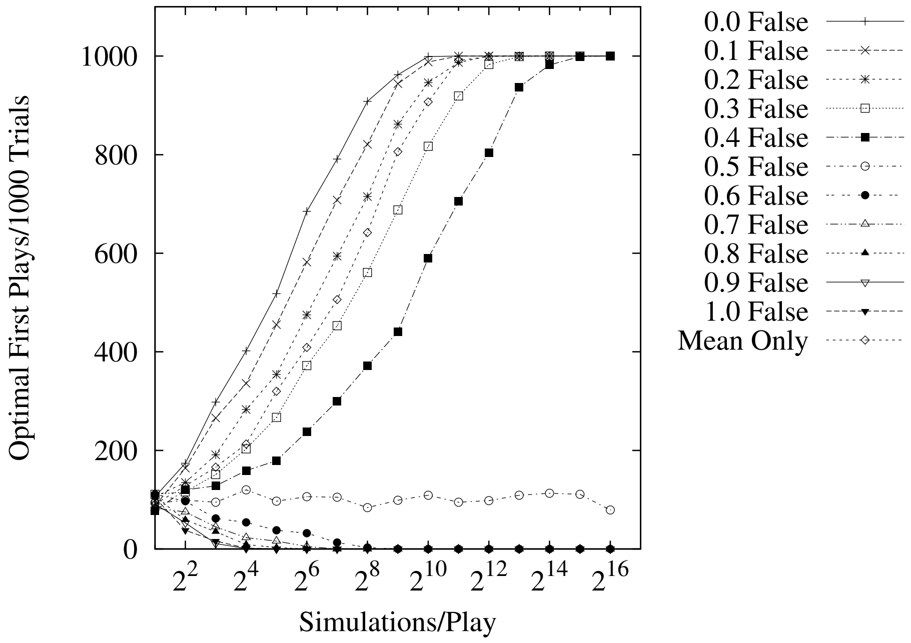


Fig. 5. Effect of False Updates in RAVE-only on SOS(10)

if we understand a little about the environment we are applying RAVE in. In games where the value of moves do not change, RAVE provides a much stronger estimate than the mean value. The false update experiments also suggest that if RAVE updates are strongly misleading, RAVE can be very detrimental, and thus, it needs to be weakened or eliminated from the estimate to improve program performance.

6 Conclusion and Future Work

The Sum of Switches game provides a simple, well-controlled environment where behaviour is easily measured. In this framework, a series of experiments with UCT and RAVE were performed. Although current trends promote parallelization as a means to increase simulations completed and program performance, the fact remains that game trees are often exponentially growing in size, meaning that simulations have to be increased by large quantities in order to produce small gains in performance. However, the RAVE experiments also suggest that by enhancing our algorithm and fine-tuning the parameters, significantly stronger play can be achieved without requiring more samples. Future work includes further investigation of RAVE in hostile environments as well as exploration on how to moderate the influence of RAVE to adapt to the environment it is in. The goal is to adapt automatically a complex UCT-based algorithm to a particular game situation.

Acknowledgements

This research was supported by the DARPA GALE project, contract No. HR0011-08-C-0110, and by NSERC, the Natural Sciences and Engineering Research Council of Canada.

References

1. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
2. van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.): CG 2008. LNCS, vol. 5131. Springer, Heidelberg (2008)
3. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go, Technical Report RR-6062 (2006)
4. Finnsson, H., Björnsson, Y.: Simulation-Based Approach to General Game Playing. In: Fox, D., Gomes, C.P. (eds.) AAAI, pp. 259–264. AAAI Press, Menlo Park (2008)
5. Lorentz, R.J.: Amazons Discover Monte-Carlo. In: [2], pp. 13–24
6. Schaeffer, J.: The History Heuristic and Alpha-Beta Search Enhancements in Practice. IEEE Trans. Pattern Anal. Mach. Intell. 11(11), 1203–1212 (1989)
7. Brügmann, B.: Monte Carlo Go (March 1993) (unpublished manuscript), http://www.cgl.ucsf.edu/go/Programs/Gobble.html
8. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Ghahramani, Z. (ed.) ICML. ACM International Conference Proceeding Series, vol. 227, pp. 273–280. ACM, New York (2007)
9. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, J., Iida, H., Heinz, E. (eds.) Advances in Computer Games. Many Games, Many Challenges. Proceedings of the ICGA / IFIP SG16 10th Advances in Computer Games Conference, pp. 159–174. Kluwer Academic Publishers, Dordrecht (2004)
10. Coulom, R.: Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength. In: [2], pp. 113–124
11. Enzenberger, M., Müller, M.: Fuego (2008), http://fuego.sf.net/ (Retrieved December 22, 2008)
12. Smith, S.J.J., Nau, D.S.: An Analysis of Forward Pruning. In: AAAI 1994: Proceedings of the Twelfth National Conference on Artificial Intelligence, Menlo Park, CA, USA, vol. 2, pp. 1386–1391. American Association for Artificial Intelligence (1994)

Creating an Upper-Confidence-Tree Program for Havannah

Fabien Teytaud and Olivier Teytaud

TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France
`fteytaud@lri.fr`

Abstract. Monte-Carlo Tree Search and Upper Confidence Bounds provided huge improvements in computer-Go. In this paper, we test the generality of the approach by experimenting on the game, Havannah, which is known for being especially difficult for computers. We show that the same results hold, with slight differences related to the absence of clearly known patterns for the game of Havannah, in spite of the fact that Havannah is more related to connection games like Hex than to territory games like Go.

1 Introduction

This introduction is based on [1]. Havannah is a 2-players board game (black vs white) invented by Christian Freeling [1,2]. It is played on an hexagonal board of hexagonal locations, with variable size (10 hexes per side usually for strong players).

White starts, after which moves alternate. The rules are straightforward.

- Each player places one stone on one empty cell. If there is no empty cell and if no player has won yet, the game is a draw (very rare cases).
- A player wins if he realizes:
  - a ring, i.e., a loop around one or more cells (empty or not, occupied by black or white stones);
  - a bridge, i.e., a continuous string of stones from one of the six corner cells to another of the six corner cells;
  - a fork, i.e., a connection between three edges of the board (corner points are not edges).

These figures are presented in Fig. 1 for a visual clarification.

Although computers can play some abstract strategy games better than any human, the best Havannah playing software plays weakly compared to human experts. In 2002, Freeling offered a prize of 1000 euros, available through 2012, for any computer program that could beat him in even one game of a ten-game match.

Havannah is somewhat related to Go and Hex. It is fully observable and involves connections; also, rings are somewhat related to the concept of an eye or a capture in the game of Go. Go has been now for a while a main target for AI

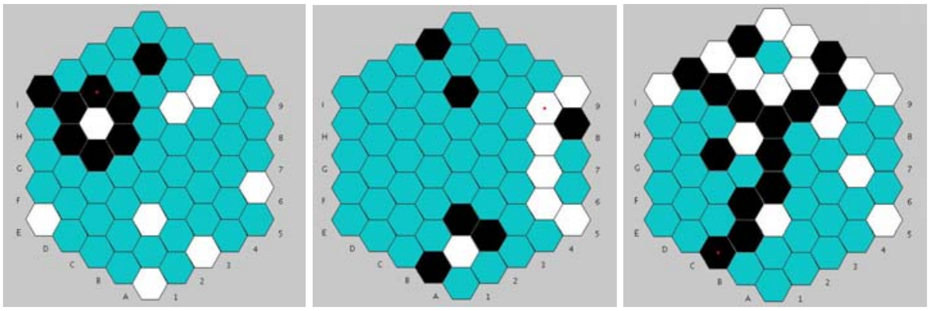


Fig. 1. Three finished games: a ring (a loop, by black), a bridge (linking two corners, by white) and a fork (linking three edges, by black)

in games (after the chess period), as it is a well-known game, and important in many Asian countries. However, it is no longer true that computers are only at the level of a novice. Since MCTS/UCT (Monte-Carlo Tree Search, Upper Confidence Trees) approaches have been defined [3,4,5], several improvements have appeared like First-Play Urgency [6], Rave-values [7,8], patterns and progressive widening [9,10], better than UCB-like (Upper Confidence Bounds) exploration terms [11], large-scale parallelization [12,13,14,15], automatic building of huge opening books [16]. Thanks to all these improvements, MoGo has already won games against a professional player in 9x9 (Amsterdam, 2007; Paris, 2008; Taiwan 2009), and recently won with handicap 6 against a professional player, Li-Chen Chien (Tainan, 2009), and with handicap 7 against a top professional player, Zhou Junxun, winner of the LG-Cup 2007 (Tainan, 2009). So, it may not be a surprise that a new game emerged. The following features make Havannah quite difficult for computers, perhaps yet more difficult than the game of Go.

- few local patterns are known for Havannah;
- no natural evaluation function;
- no pruning rule for reducing the number of reasonable moves;
- large action space (271 for the first move with board size 10).

The advantage of Havannah, similar as in Go (except in pathological cases for the game of Go) is that simulations have a bounded length: the size of the board.

The goal of this paper is to investigate the generality of the MCTS/UCT approach by testing it in Havannah.

By way of notation, $x \pm y$ means a result with average x, and 95% confidence interval $[x - y, x + y]$ (*i.e.,* y is two standard deviations).

2 UCT

Upper Confidence Trees are the most straightforward choice when implementing a Monte-Carlo Tree Search. The basic principle is as follows. As long as there is time before playing, the algorithm performs random simulations from a UCT tree leaf. These simulations (playouts) are complete possible games, from the root of the tree until the game is over, played by a random player playing both black and white. In its most simple version, the random player, which is used when in a situation s which is not in the tree, just plays randomly and uniformly among legal moves in s; we did not implement anything more sophisticated, as we are more interested in algorithms than in heuristic tricks. When the random player is in a situation s already in the UCT tree, then its choices depend on the statistics: number of wins and number of losses in previous games, for each legal move in s. The detailed formula used for choosing a move depending on statistics is termed a bandit formula, discussed below. After each simulation, the first situation of the simulated game that was not yet in memory is archived in memory, and all statistics of numbers of wins and numbers of defeats are updated in each situation in memory which was traversed by the simulation.

Bandit Formula

The most classical bandit formula is UCB (Upper Confidence Bounds [17,18]); Monte-Carlo Tree Search based on UCB is termed UCT. The idea is to compute a exploration/exploitation score for each legal move in a situation s for choosing which of these moves must be simulated. Then, the move with maximal exploration/exploitation score is chosen. The exploration/exploitation score of a move d is typically the sum of the empirical success rate $score(d)$ of the move d, and of an exploration term which is strong for less explored moves. UCB uses Hoeffding's bound for defining the exploration term:

$$exploration_{Hoeffding} = \sqrt{\log(2/\delta)/n}, \tag{1}$$

where n is the number of simulations for move d. δ is usually chosen linear as a function of the number m of simulations of the situation s: δ is linear in m. In our implementation, with a uniform random player, the formula was empirically set to:

$$exploration_{Hoeffding} = \sqrt{0.25\log(2+m)/n}. \tag{2}$$

[19,20] have shown the efficiency of using Bernstein's bound instead of Hoeffding's bound, in some settings. The exploration term is then:

$$exploration_{Bernstein} = \sqrt{score(d)(1-score(d))2\log(3/\delta)/n} + 3\log(3/\delta)/n \tag{3}$$

This term is smaller for moves with small variance ($score(d)$ small or large).

After the empirical tuning of Hoeffding's equation 2, we tested Bernstein's formula as follows (with $p = score(d)$ for short)

$$exploration_{Bernstein} = \sqrt{4Kp(1-p)\log(2+m)/n} + 3\sqrt{2}K\log(2+m)/n. \tag{4}$$

Within the "2+", which is here for avoiding special cases for 0, this is Bernstein's formula for δ linear as a function of m.

We tested several values of K. The first value 0.250 corresponds to Hoeffding's bound except that the second term is added. The results are given in Table 1. Experiments are performed with 1000 simulations per move, with size 5. We tried to experiment values below 0.01 for K but with poor results.

Table 1. Results of testing the K values

K	Score against Hoeffding's formula 2
0.250	0.503 ± 0.011
0.100	0.578 ± 0.010
0.030	0.646 ± 0.005
0.010	0.652 ± 0.006
0.001	0.582 ± 0.012
0.000	0.439 ± 0.015

Scaling of UCT

Usually, UCT provides increasingly better results when the number of simulations per move is growing. Typically, the winning rate with $2z$ simulations against UCT with z simulations is roughly 63% in the game of Go (see, e.g., [12]). In the case of Havannah, with a uniform random player (i.e., the random player plays uniformly among legal moves) and the bandit formula as in Eq. 2, we obtain the results given in Table 2 for the UCT tuned as above ($K = 0.25$). These experiments are performed on a board of size 8.

Table 2. Results for UCT

Number of simulations of both player	Success rate
250 vs 125	0.75 ± 0.02
500 vs 250	0.84 ± 0.02
1000 vs 500	0.72 ± 0.03
2000 vs 1000	0.75 ± 0.02
4000 vs 2000	0.74 ± 0.05

3 Guiding Exploration

UCT-like algorithms are quite strong for balancing exploration and exploitation. As against it, they provide no information for unexplored moves, and on how to choose among these moves; and little information for loosely explored moves. Various tricks have been to overcome this handicap. We mention three of them: First Play Urgency, Progressive widening, Rapid Action Value Estimates (RAVE). The last two one discussed below.

3.1 Progressive Widening/Unpruning

In progressive widening [9,10,21], we first rank the legal moves at a situation s according to some heuristic: the moves are then renamed 1, 2, $\dots$,n, with $i < j$ if move i is preferred to move j by the heuristic. Then, at the m^{th} simulation of a node, all moves with index larger than $f(m)$ have $-\infty$ score (i.e., are discarded), with $f(m)$ some non-decreasing mapping from $\mathbb{N}$ to $\mathbb{N}$. It was shown in [21] that this can work even with random ranking, with $f(m) = \lfloor Km^{1/4} \rfloor$ for some constant K. In [9] it was shown that $f(m) = \lfloor Km^{1/3.4} \rfloor$ for some constant K performs well in the case of Go, with a pattern-based heuristic. The algorithm proposed in [10] and now used also in MoGo is a bit different: an exploration term depending on a pattern-based heuristic and decreasing logarithmically with the number of simulations of this move is added to the score; for move d in situation s,

$$newScore(d,s) = score(d,s) + H(d,s)/\log(2+m),$$

where as previously n is the number of simulations including move d at situation s. In the case of Havannah, we have no such heuristic. We decided to use heuristic-free progressive widening, as it was shown in [21] that an improvement can be provided even if no heuristic is available (i.e., the order is arbitrary). This idea of using progressive widening without heuristic is a bit counter-intuitive. However, consider the following case. Consider a node of a tree which is explored 50 times only (this certainly happens for many nodes deep in the tree). If there are 50 legal moves at this node, then the 50 simulations will be distributed on the 50 legal moves (one simulation for each legal move), if you use the standard UCB formula instead of (without) progressive widening. Meanwhile, progressive widening will sample a few moves only, e.g., 4 moves, and sample much more the best of these 4 moves - this is likely to be better than taking the average of all moves as an evaluation function. We experimented with 500 simulations per move, size 5, various constants P and Q for the progressive widening $f(m) = Q\lfloor m^P \rfloor$ (see Table 3). These experiments were performed with the exploration formula given in Eq. 2. We tested various other parameters for Q and P, without seeing a significant improvement. Perhaps improvements are possible by jointly tuning the Hoeffding's bound and the progressive widening formula.

Table 3. Results of progressive widening

Q, P	Success rate against no prog. widening
1, 0.7	0,496986 ± 0,014942
1, 0.8	0.51 ± 0,0235833
1, 0.9	0.50 ± 0,0145172
4, 0.4	0,500454 ± 0,0134803
4, 0.7	0.49 ± 0,0181818
4, 0.9	0,485101 ± 0,0172619

3.2 Rapid Action Value Estimate

In the case of Go, [7,8] propose to average the score with a permutation-based statistical estimate. Precisely, the score for a move d in situation s simulated n times becomes:

$$newScore(d,s) = (1-\alpha(n))score(d,s) + \alpha(n)rave(d,s) + exploration$$

where

- $score(d, s)$ is the ratio of won simulations among simulations with situation s and move d in s, $rave(d, s)$ is the proportion of won games among games *containing* move d after situation s;
- $\alpha(n) \to 0$ as $n \to \infty$, whereas $\alpha(n)$ is close to 1 for n small, so that the heuristic *rave* values are used initially, and eventually they are replaced by "real" values.

The difference between $score(d, s)$ and $rave(d, s)$ is that the proportion of won games in $rave()$ is computed among all simulations *containing d as move after situation s* and not only all simulations *with d as move in situation s*. In the game of GO, RAVE values are a great improvement. However, they involve complicated implementations due to captures and re-captures. In the case of Havannah there is no such problem, and we will see that the results are good. We used the following formula:

$$\alpha(n) = R/(R+n), \quad exploration = exploration_{Hoeffding} = \sqrt{K \log(2+m)/n}.$$

R was empirically set to 50 (on games with size 5 with 1000 simulations per move); intuitively, R is the number of simulations before the weight of "RAVE" values is equal to the weight of "greedy" values (see Table 4). All scores are against UCT with Eq. 2. $K = 0$ is then the best constant. This was also pointed out in [11] for the game of GO. We then tested larger numbers of simulations, i.e., 30,000 simulations per move. Disappointingly but consistently with [11], we had to change the coefficients in order to obtain positive results, whereas the tuning of UCT is seemingly more independent of the number of simulations per move (see Table 5). The first line corresponds to the configuration empirically chosen for 1000 simulations per move (see Table 4); its results are disappointing, almost equivalent to UCT, for these experiments with 30,000 simulations per move. The second line uses the same exploration constant as UCT, but it is seemingly too much. Then, in the following lines, using a weaker exploration and a small value of R, we obtain better results. [11] points out that, with big simulation times, $K = 0$ was better, but an exploration bonus depending on patterns was used instead.

As a conclusion, for large numbers of simulations, RAVE is not as efficient as for small values (when compared to UCT). Perhaps better tuning could yield better results. The main weaknesses of RAVE are:

- tuning is required for each new number of simulations;
- the results for large numbers of simulations are less impressive (yet significant).

Table 4. Results of RAVE (1000 sim.)

size 5	R=50	K=0.25	size 5	1000 simulations/move	47.26% ± 4.0 %
	R=50	K=0.05	size 5	1000 simulations/move	60.46% ± 2.9 %
	R=50	K=0	size 5	1000 simulations/move	95.33% ± 0.01 %
size 8	R=50	K=0	size 8	1000 simulations/move	100% on 1347 runs

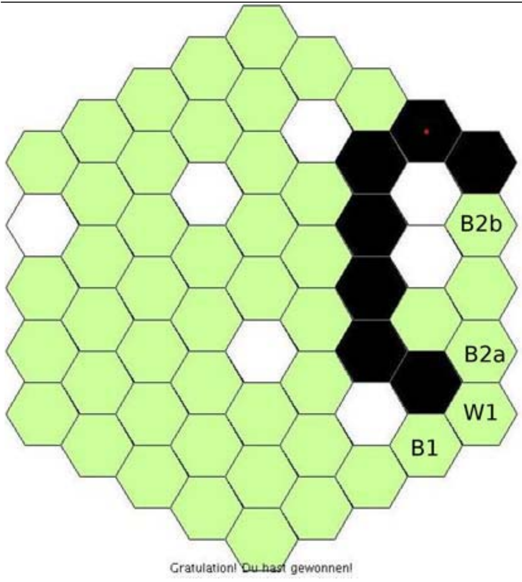
Table 5. Results of RAVE (30,000 sim.)

R=50	K=0	size 5	30,000 simulations/move	0.53 ± 0.02
R=50	K=0.25	size 5	30,000 simulations/move	0.47 ± 0.04
R=50	K=0.05	size 5	30,000 simulations/move	0.60 ± 0.02
R=50	K=0.02	size 5	30,000 simulations/move	0.60 ± 0.03
R=5	K=0.02	size 5	30,000 simulations/move	0.61 ± 0.06
R=20	K=0.02	size 5	30,000 simulations/move	0.66 ± 0.03

In contrast, the efficiency increases with the size of the action space - this is promising for the application of RAVE to large action spaces.

4 Games against Havannah-Applet

We tested our program against Havannah-Applet http://dfa.imn.htwk-leipzig.de/havannah/, recommended by the MindSports association as the



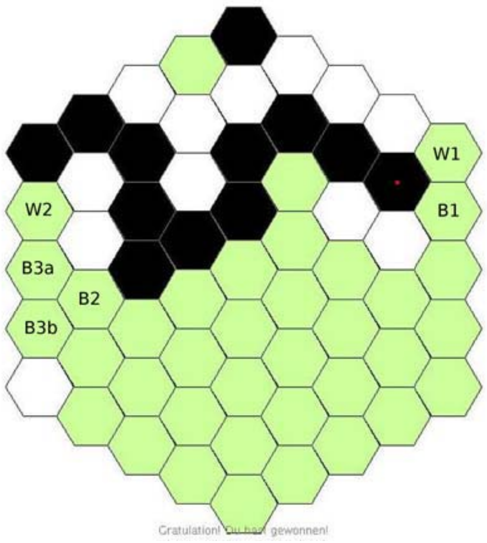
Success rate estimated by Havannah-Applet	Success rate estimated by our software
62.6%	46.4%
60.8%	47.2%
64.0%	51.2%
56.5%	54.8%
62.5%	65.4%
57.3%	63.7%
71.0%	88.8%
0.5%	

Fig. 2. Left: the result of the game played against Havannah-Applet in size 5. Our program won rather quickly, by a nice multiple constraint: the white opponent must play *W1* (unless black wins by bridge). Then, black can connect to the lower-right edge with *B1*. Then, Black has two opportunities for connecting to the right edge, *B2a* and *B2b*; White can only block one and Black then realizes a fork. As seen on the estimated success rate, Havannah-Applet did not suspect this attack before the very last move. Right: estimated success rate for each of the opponents.

only publicly available program that plays by the Havannah rules. There are 30 seconds per move, but we restricted our program to running in 8 seconds per move (first game) and 2.5 seconds per move (second game). In both cases, our program, based on RAVE, no exploration term, no progressive widening, was Black (White starts and has therefore the advantage in Havannah). The first game (played with 8 seconds per move for our program, against 30s for the opponent) is presented in Fig. 2.

Consistently, the estimated success rate is lower than 50 % at the beginning (as the opponent, playing first, has the advantage initially). It then increases regularly until the end. The second game is presented in Fig. 3, with only 7000 simulations (nearly 2 seconds) per move.

Consistently again, the estimated success rate is lower than 50 % at the beginning (as the opponent, playing first, has the advantage initially). It then increases regularly until the end.



Success rate estimated by Havannah-Applet	Success rate estimated by our software
62.3%	45.7%
60.3%	48.2%
50.6%	49.9%
43.0%	55.9%
40.4%	52.6%
52.6%	55.4%
40.7%	56.2%
42.0%	63.5%
31.1%	60.0%
42.2%	75.8%
36.8%	68.6%
27.4%	78.1%
3.7%	

Fig. 3. Left: the result of the game played against Havannah-Applet in size 5. Our program was playing black and won by resignation. White has to play *W1*, otherwise Black realises a bridge. Then, Black plays *B1* and is connected to a second side. Next, White must play *W2* (if Black plays *W2* then Black has a fork). Finally, Black can play *B2* and with this move, it is connected to the third side (by *B3a* or *B3b* (White cannot avoid this connection). Right: estimated success rate for each of the opponents.

5 Discussion

In the case of Havannah we could clearly validate the efficiency of some well known techniques coming from computer-Go, showing the generality of the MCTS approach. Essentially they are as follows.

- The efficiency of Bernstein's formula, in front of Hoeffding's formula, is clear (up to 65%).
- The continuous success rate of UCT with $2k$ simulations per move, against UCT with k simulations per move, nearly holds in the case of Havannah (nearly 75%, whereas it is usually around 63% for MCTS in the game of Go [12]). However, the success rate is higher than in the case of the game of Go (around 75%).
- The efficiency of the RAVE heuristic is clearly validated. The main strength is that the efficiency increases with the size, reaching 100 % on 1347 games in size 8. In contrast, RAVE becomes less efficient, and requires tuning, when the number of simulations per move increases - however, we still keep 66% of success rate).
- Progressive widening, in spite of the fact that it was shown in [21] that it works even without heuristic, was not significant for us. In the case of Go, progressive widening was shown quite efficient in implementations based on patterns [9,10].
- Our program could defeat Havannah-Applet easily, whereas it was playing as Black, and with only 2s per move instead of 30s (running on a single core). Running more experiments was difficult due to lack of automated interface.

References

1. Wikipedia, Havannah (2009)
2. Schmittberger, R.W.: New Rules for Classic Games. Wiley, Chichester (1992)
3. Chaslot, G., Saito, J.T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo Strategies for Computer Go. In: Schobbens, P.Y., Vanhoof, W., Schwanen, G. (eds.) Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium, pp. 83–91 (2006)
4. Coulom, R.: Efficient selectivity and backup operators in monte-carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2007)
5. Kocsis, L., Szepesvari, C.: Bandit-based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
6. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii, pp. 175–182 (2007)
7. Bruegmann, B.: Monte carlo go (1993) (Unpublished)
8. Gelly, S., Silver, D.: Combining online and offline knowledge in uct. In: ICML 2007: Proceedings of the 24th international conference on Machine learning, New York, NY, USA, pp. 273–280. ACM Press, New York (2007)
9. Coulom, R.: Computing elo ratings of move patterns in the game of go. In: Computer Games Workshop, Amsterdam, The Netherlands (2007)
10. Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., Bouzy, B.: Progressive strategies for monte-carlo tree search. In: Wang, P. (ed.) Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007), pp. 655–661. World Scientific Publishing Co. Pte. Ltd., Singapore (2007)

11. Lee, C.S., Wang, M.H., Chaslot, G., Hoock, J.B., Rimmel, A., Teytaud, O., Tsai, S.R., Hsu, S.C., Hong, T.P.: The computational intelligence of mogo revealed in taiwan's computer go tournaments. IEEE Transactions on Computational Intelligence and AI in Games (2009) (accepted)
12. Gelly, S., Hoock, J.B., Rimmel, A., Teytaud, O., Kalemkarian, Y.: The parallelization of monte-carlo planning. In: Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008), pp. 198–203 (2008) (to appear)
13. Chaslot, G., Winands, M., van den Herik, H.: Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131. Springer, Heidelberg (2008)
14. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Proceedings of CGW 2007, pp. 93–101 (2007)
15. Kato, H., Takeuchi, I.: Parallel monte-carlo tree search with simulation servers. In: 13th Game Programming Workshop, GPW 2008 (November 2008)
16. Audouard, P., Chaslot, G., Hoock, J.B., Perez, J., Rimmel, A., Teytaud, O.: Grid coevolution for adaptive simulations; application to the building of opening books in the game of go. In: Proceedings of EvoGames (2009)
17. Lai, T., Robbins, H.: Asymptotically efficient adaptive allocation rules. Advances in Applied Mathematics 6, 4–22 (1985)
18. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite time analysis of the multiarmed bandit problem. Machine Learning 47(2/3), 235–256 (2002)
19. Audibert, J.Y., Munos, R., Szepesvari, C.: Use of variance estimation in the multiarmed bandit problem. In: NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation (2006)
20. Mnih, V., Szepesvári, C., Audibert, J.Y.: Empirical Bernstein stopping. In: ICML 2008: Proceedings of the 25th international conference on Machine learning, New York, NY, USA, pp. 672–679. ACM, New York (2008)
21. Wang, Y., Audibert, J.Y., Munos, R.: Algorithms for infinitely many-armed bandits. In: Advances in Neural Information Processing Systems., vol. 21 (2008)

Randomized Parallel Proof-Number Search

Jahn-Takeshi Saito[1], Mark H.M. Winands[1], and H. Jaap van den Herik[2]

[1] Department of Knowledge Engineering
Faculty of Humanities and Sciences, Maastricht University
{j.saito,m.winands}@maastrichtuniversity.nl
[2] Tilburg centre for Creative Computing (TiCC)
Faculty of Humanities, Tilburg University
h.j.vdnherik@uvt.nl

Abstract. Proof-Number Search (PNS) is a powerful method for solving games and game positions. Over the years, the research on PNS has steadily produced new insights and techniques. With multi-core processors becoming established in the recent past, the question of parallelizing PNS has gained new urgency. This article presents a new technique called Randomized Parallel Proof-Number Search (RP–PNS) for parallelizing PNS on multi-core systems with shared memory. The parallelization is based on randomizing the move selection of multiple threads, which operate on the same search tree. RP–PNS is tested on a set of complex Lines-of-Action endgame positions. Experiments show that RP–PNS scales well. Four directions for future research are given.

1 Introduction

Most computer programs for board games successfully employ $\alpha\beta$ search. For some games, however, $\alpha\beta$ search displays a weakness in the endgame that can currently neither be overcome by endgame databases nor by other $\alpha\beta$ extensions. To remedy the deficit, mate searches may be applied. One such alternative to $\alpha\beta$ search is Proof-Number Search (PNS). PNS enjoys popularity as a powerful method for solving endgame positions and complete games. Since its introduction by Allis *et al.* [1] in 1994, PNS has developed into a whole family of search algorithms (e.g., PN^2 [2], PDS [3], and df-pn [4]) with applications to many games, such as Shogi [5], the one-eye problem in Go [6], Checkers [7], and Lines of Action [8].

A variety of parallel $\alpha\beta$ algorithms have been proposed in the past [9], but so far not much research has been conducted on parallelizing PNS. With multi-core processors becoming established as standard equipment, parallelizing PNS has become an important topic. Pioneering research has been conducted by Kishimoto [10], who parallelized the depth-first PNS variant PDS. His algorithm is called ParaPDS and is designed for distributed memory systems. In this article we address the problem of parallelizing PNS and PN^2 for shared memory systems. The parallelization is based on randomizing the move selection of multiple threads, which operate on the same search tree. This method is called

Randomized Parallel Proof-Number Search (RP–PNS). Its PN^2 version is called RP–PN^2.

The article is organized as follows. Section 2 describes the PNS algorithm and two of its variants. Section 3 discusses the options for the parallelization of PNS in general terms. Section 4 introduces the new parallel PNS, RP–PNS. Section 5 shows and discusses the results of testing RP–PNS on a set of complex Lines-of-Action endgame positions. Section 6 provides a conclusion and gives four directions for future research.

2 Proof-Number Search

This section describes the sequential PNS algorithm (Subsect. 2.1) and two of its variants, PDS and PN^2 (Subsect. 2.2).

2.1 Sequential PNS

PNS is a best-first search for AND/OR trees. The search aims at proving or disproving a binary goal, i.e., a goal that can be reached by the first player or be refuted by the second player under optimal play by both sides. Each node N in the tree contains two numbers called the *proof number* ($pn(N)$) and the *disproof number* ($dn(N)$), respectively.

PNS iterates the best-first search cycle consisting of three steps.

1. Selection: starting at the root, a path P consisting of successor nodes is created until a leaf L is found; a heuristic guides the selection of successors.
2. Expansion: L is expanded and its children's proof and disproof numbers are set.
3. Back-up: the new values of the expanded node L are propagated back to the root.

Informally, the algorithm runs as follows. The selection step finds a leaf L of the tree. In PNS, L is called the *most-proving node*, i.e., the node that is expected to reach a proof (or disproof) with the fewest additional expansions. The most-proving node is found by heuristically descending a path P in the tree starting at the root. At every node N, the *best successor* ($bs(N)$) is selected and this $bs(N)$ becomes the new N. This procedure is repeated until a leaf L is reached. The best successor of N is determined differently in OR and AND nodes. (1) In OR nodes (when player 1 moves), the $bs(N)$ is the child that requires the fewest number of additional expansions to prove the goal. $pn(N)$ represents this number. (2) In AND nodes (when player 2 moves), $bs(N)$ is the child that requires the fewest additional expansions to disprove the goal. $dn(N)$ represents this number.

More formally we describe the best successor and the successor number as follows. Given a non-terminal node N, its children are denoted by $N_i, i = 1, ..., |N|$ where $|N|$ is the number of children of N. If N is an OR node, the N_i are sorted increasingly by their proof number $pn(N_i)$. If N is an AND node, the N_i are sorted increasingly by their disproof number $dn(N_i)$. The *successor number* of

Rules for *AND* nodes:

$$pn(N) = \sum_{S \in successor(N)} pn(S)$$

$$dn(N) = \min_{S \in successor(N)} dn(S)$$

Rules for *OR* nodes:

$$pn(N) = \min_{S \in successor(N)} pn(S)$$

$$dn(N) = \sum_{S \in successor(N)} dn(S)$$

Fig. 1. Rules for updating proof and disproof numbers for a node N

N for a child N_i is $sn(N_i) = pn(N_i)$ if N is an OR node and $sn(N_i) = dn(N_i)$ if N is an AND node. The best successor of N is the child N_1. The *best successor number* $bsn(N)$ is $sn(N_1)$.

The expansion step of the cycle expands L and initializes its children's proof and disproof numbers. If a new child directly proves the goal, its proof number is set to 0 and its disproof number is set to infinity. Correspondingly, if a new child directly disproves the goal, its disproof number is set to 0 and its proof number is set to infinity. If the child neither proves nor disproves, the number of children of the leaf can be used to set these numbers.[1]

In the back-up step the newly assigned proof and disproof numbers are propagated back to the root changing the proof and disproof number in each node on the path P. The rules applied for updating proof and disproof numbers are given in Fig. 1.

After the root has been reached and its values have been updated, the cycle is complete. The cycle is repeated until the termination criterion is met. The criterion is satisfied if either the root's proof number is 0 and the disproof number is infinity, or vice versa. In the first case, the goal is proved. In the second case the goal is disproved.

Figure 2 (placed in Sect. 4 where it is also used for explanation) shows a search tree with proof and disproof numbers. The proof and disproof numbers of the interior nodes can be calculated from the children's numbers by applying the updating rules presented in Fig. 1.

2.2 PDS and PN^2

A weakness of PNS is its memory consumption. This problem arises because the whole tree is stored in memory. There are many variants of PNS that address the memory problem; two of them are PDS and PN^2.

PDS by Nagai [3] solves the memory problem by transforming the best-first search into a depth-first search. PDS applies multiple-iterative deepening at every node. PDS can only function successfully by means of a transposition table to speed up the re-searches.

Like PDS, PN^2 [1] reduces memory requirements of PNS by re-searching parts of the tree. PN^2 consists of two levels of PNS. The first level PNS (PN$_1$) calls

[1] We remark that other methods for estimating the proof and disproof numbers of newly expanded leafs have been proposed [11].

a PNS at the second level (PN_2) for an evaluation of the most-proving node of the PN_1-search tree. This PN_1 search is bound by a maximum number of nodes M to be stored in memory. Different ways have been proposed to set this bound [1,11]. The PN_1 search is stopped when the number of nodes stored in memory exceeds M or the subtree is (dis)proved. After completion of the PN_1 search, the children of the root of the PN_1-search tree are preserved, but subtrees are removed from memory.

3 Parallelization of PNS

This section introduces some basic concepts for describing the behavior of parallel search algorithms (Subsect. 3.1), outlines ParaPDS, a parallelization of PDS (Subsect. 3.2), and explains parallel randomized search (Subsect. 3.3).

3.1 Terminology

Parallelization aims at reducing the time that a sequential algorithm requires for successful terminating. The speedup is achieved by distributing computations to multiple threads executed in parallel.

Parallelization gains from dividing computation over multiple resources but simultaneously this may impose a computational cost. According to Brockington and Schaeffer [12] three kinds of overhead may occur when parallelizing a search algorithm: (1) search overhead, resulting from extra search not performed by the sequential algorithm; (2) synchronization overhead, created at synchronization points when one thread is idle waiting for another thread; (3) communication overhead, created by exchanging information between threads.

Search overhead is straightforward and can be measured by the number of additional nodes searched. Synchronization and communication overhead depend on the kind of information sharing used. There are two kinds of information sharing: (i) message passing and (ii) shared memory. Message passing simply consists of passing information between memory units exclusively accessed by a particular thread. Under shared memory all threads can access a common part of memory. With the advent of multi-core CPUs memory sharing has become common place.

An important property governing the behavior of parallel algorithms is scaling. It describes the efficiency of parallelization with respect to the number of threads as a fractional relation t_1/t_T between the time t_1 for terminating successfully with one thread and the time t_T for terminating successfully with T threads.

3.2 ParaPDS and the Master-Servant Design

The only existing parallelization of PNS described in the literature has so far been ParaPDS by Kishimoto and Kotani [10].[2] This pioneering work of parallel PNS achieved a speedup of 3.6 on 16 processors on a distributed memory

[2] Conceptually related to PNS is Conspiracy Number Search (CNS) by McAllester [13]. Lorenz [14] proposes to parallelize a variant of CNS (PCCNS). PCCNS uses a master-servant model ("Employer-Worker Relationship", ibid.).

machine. Therefore, processes are used instead of threads for parallelization. ParaPDS relies on a master-servant design. One master process is coordinating the work of several servant processes. The master manages a search tree up to a fixed depth *d*. The master traverses through the tree in a depth-first manner typical for PDS. On reaching depth *d* it assigns the work of searching further to an idle servant. The search results of the servant process are backed up by the master.

The overhead created by ParaPDS is mainly a search overhead. There are two reasons for this overhead: (1) lack of a shared-memory transposition table, and (2) the particular master-servant design. Regarding reason (1), ParaPDS is asynchronous, i.e., no data is passed between the processes except at the initialization and the return of a servant process. ParaPDS thereby avoids message passing. The algorithm is designed for distributed-memory machines common at the time ParaPDS was invented (i.e., 1999). Transposition tables are important to PDS, as this variation of PNS performs iterative deepening. An implication of using distributed-memory machines is that ParaPDS cannot profit from a shared transposition table and loses time on re-searching nodes. Regarding reason (2), the master-servant design can lead to situations in which multiple servant processes are idle because the master process is too busy updating the results of other processes or finding the next candidate to pass to a servant process.

One may speculate that the lack of a shared-memory transposition table in ParaPDS could nowadays be amended to a certain degree, at the expense of a synchronization overhead, by the availability of shared-memory machines. However, the second reason for the overhead of the master-servant design still remains.

3.3 Randomized Parallelization

An alternative to the master-servant design of ParaPDS for parallelizing tree-search is *randomized parallelization*. Shoham and Toledo [15] proposed the method for parallelizing *any* kind of best-first search on AND/OR trees. The method relies on a heuristic which may seem counterintuitive at first. Instead of selecting the child with the best heuristic evaluation, a probability distribution of the children determines which node is selected. Shoham and Toledo call this a *randomization* of the move selection. Randomized Parallel Proof-Number Search (RP–PNS) as proposed in this contribution adheres to the principle of randomized parallelization. The specific probability distribution is obviously based on the selection heuristic.

The master-servant design of ParaPDS and randomized parallelization may be compared as follows. ParaPDS maintains a privileged master thread; only the master thread operates on the top level tree; the master thread selects the subtree in which the servant threads search; it also coordinates the results of the servant processes; each servant thread maintains a separate transposition table in memory. In randomized parallelization there is no master thread; each thread is guided by its own probabilities for selecting the branch to explore; there is no communication overhead but instead there is synchronization overhead; all threads can operate on the same tree which is held in shared memory.

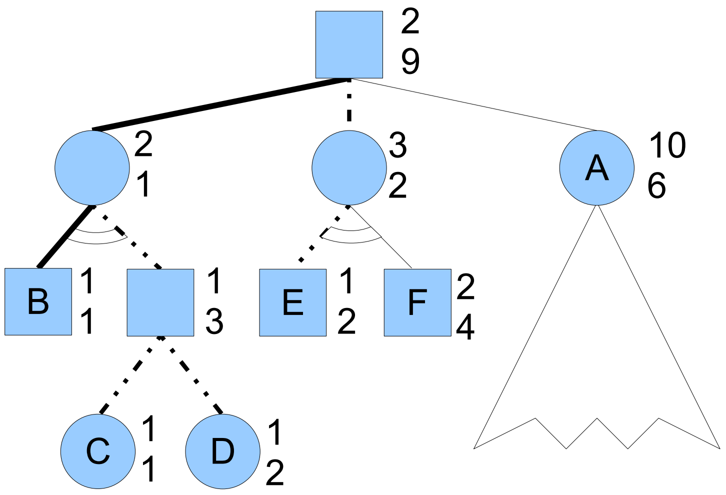


Fig. 2. Example of a PNS tree. Squares represent OR nodes; circles represent AND nodes. Depicted next to each node are its proof number at the top and its disproof number at the bottom.

4 RP–PNS

This section introduces RP–PNS. Subsection 4.1 explains the basic functioning of RP–PNS and describes how it differs from ParaPDS. Subsection 4.2 explains details of an implementation of RP–PNS.

4.1 Detailed Description of Randomized Parallelization for PNS

There are two kinds of threads in RP–PNS: (1) principal-variation (PV) threads, and (2) alternative threads. RP–PNS maintains one PV thread; all other threads operating on the search tree are alternative threads.

The PV thread always applies the same selection strategy as sequential PNS. It thereby operates on the PV, i.e., the path from root to leaf following the heuristic for finding the most-proving node. We call this selection strategy *PV selection strategy* and a child on the PV, a *PV node*.

The alternative threads select a node according to a modified selection strategy. Instead of minimizing the successor number, there is a chance that a suboptimal successor number is accepted. A probability distribution in the heuristic creates the desired effect: the expanded nodes are always close to the PV since nodes expanded in alternative threads would likely be on the PV at a later cycle. The alternative threads anticipate a possible future PV. The probability of a suboptimal node to be selected for an alternative thread depends on the degree by which it deviates from the PV. In the selection step, alternative threads consider only a subset of all children. The considered children have a successor number at most D larger than the best successor number. An alternative thread selects one of these children by a certain probability.

To account for the move selection more formally, we first introduce further notation. Similarly, for some positive natural number c, we can count the children N_i with successor number $sn(N)$ smaller than or equal to some positive integer c. This count is $cnt(N, c) = |\{N_i : bsn(N_i) \leq c, \text{for } i = 1, ..., |N|\}|$.

Let T be the number of threads used. For each thread $\theta_t, t = 1, ..., T$ and node N there is a probability distribution $P_{\theta_t,N}$ that assigns a probability $p(\theta_t, N, N_i)$ to each child N_i of N. This is the probability of node N_i to be selected as successor node. Equation 1 defines the probability for selecting N_i at N. θ_1 is the PV thread.

$$p(\theta_t, N_i) = \begin{cases} 0 & : \text{ if } \quad t = 1 \wedge sn(N_i) > min(N) \\ cnt(N, min(N))^{-1} & : \text{ if } \quad t = 1 \wedge sn(N_i) = min(N) \\ 0 & : \text{ if } \quad t \neq 1 \wedge sn(N_i) > min(N) + D \\ cnt(N, min(N) + D)^{-1} & : \text{ if } \quad t \neq 1 \wedge sn(N_i) \leq min(N) + D \end{cases} \tag{1}$$

The parameter D in Equation 1 regulates the degree to which the alternative threads differ from the PV. Setting $D = 0$ will result in the PV selection strategy.[3] Setting D too high results in threads straying too far from the PV.

Figure 2 illustrates the consequences of varying the parameter D. In this example, the PV is represented by the bold line and reaches leaf B. An alternative selection with $D = 1$ is represented by the bold, dotted line. It will select one of the leafs B, C, D, or E with equal probability. Setting $D = 2$ will result in also selecting F. We note that the subtree at A is selected only for $D \geq 8$.

In addition to these probabilities, for all alternative threads we assign a second probability of deviating from the PV. This is done by choosing with a probability of $2/d$ randomly from N_2 and N_3 (determined by trial-and-error) instead of choosing N_1 if so far the thread has not deviated from the PV. This choice is determined by the depth d of the last PV. The additional randomization is necessary because it enables sufficient deviation from the PV in case that D is not large enough to produce any effect.

We remark that RP–PNS differs from the original randomized parallelization with respect to three points. (1) Shoham and Toledo do not distinguish between PV and alternative threads. (2) The original randomized parallelization selects children with a probability proportionate to their best-first value while RP–PNS uses an equidistribution for the best candidates. (3) The original randomized parallelization does not rely on a second probability. The differences in point 2 and 3 are based on the desire to produce more deviations from the PV in order to avoid that too many threads congest the same subtree. The selection in RP–PNS is similar to Buro's selection of a move from an opening book [16].

[3] More precisely, this is true if there exists exactly one child N_i with $sp(N_i) = bsn(N)$. If multiple children have the same best successor number, the alternative threads can deviate from the PV which we assume to be selected deterministically in PNS.

In RP–PNS multiple threads operate on the same tree. To facilitate the parallel access some complications in the implementation require our attention. The next subsection gives details of the actual implementation of RP–PNS.

4.2 Implementation

As pointed out in the previous subsection, all threads in RP–PNS operate on the same search tree held in shared memory. In order to prevent errors in the search tree, RP–PNS has to synchronize the threads. This is achieved in the implementation by a locking policy. Each tree node has a lock. It guarantees that only one thread at a time operates on the same node while avoiding deadlocks. The locking policy consists of two parts: (1) when a thread selects a node, it has to lock it; (2) when a thread updates a node N it has to lock N and its parent. The new values for N are computed. After N has been updated, it is released and the updating continues with the parent.

Each node N maintains a set of flags, one for each thread, to facilitate the deletion of subtrees. Each flag indicates whether the corresponding thread is in the subtree below N. A thread can delete the subtree of N only if no other thread has set its flag in N.

If a transposition table is used to store proof and disproof numbers, each table entry needs an additional lock. The number of locks for the transposition table could be reduced by sharing locks for multiple entries. Similar policies have been used in parallel Monte-Carlo Tree Search [17] (to which the master-servant design has also been applied [18]).

Synchronization imposes a cost on RP–PNS in terms of memory and time consumption. The memory consumption increases due to the additional locks (per node, 16 bytes for a spinlock and flags, cf. [17]) in each node.

The overhead is partially a synchronization overhead and partially a search overhead. The synchronization overhead occurs whenever a thread has to wait for another thread due to the locking policy or due to the transposition table locking. The search overhead is created by any path that would not have been selected by the sequential PNS and that at the same time does not contribute to find the proof. The following section describes experiments that also test the overhead of RP–PNS.

5 Experiment

This section presents experiments and results for RP–PNS. Subsection 5.1 outlines the experimental setup, Subsect. 5.2 shows the results obtained, and Subsect. 5.3 discusses the findings.

5.1 Setup

We implemented RP–PNS as described in the previous section and tested it on complex endgame positions of Lines of Action (LOA)[4]. We chose LOA because

[4] The test set is available at `http://www.personeel.unimaas.nl/m-winands/loa/`, "Set of 286 hard positions."

it is an established domain for applying PNS. The test set consisting of 286 problems has been applied before frequently [19,20,8].

The experiment tests two parallelization methods: RP–PNS and RP–PN^2 (an adaptation of RP–PNS for PN^2) for 1, 2, 4, and 8 threads. The combination of an algorithm with a specific number of threads is called a *configuration* and denoted by indexing the number of threads, e.g., RP–PNS$_8$ is RP–PNS using eight threads. We remark that PNS = RP–PNS$_1$ and RP–PN^2 = RP–PN$^2_1$.

The implementation of RP–PN^2 uses RP–PNS for PN$_1$ and PNS for PN$_2$. The size of PN$_2$ was limited to S^ϵ/T, where S is the size of the PN$_1$ tree, T is the number of threads used, and ϵ is a parameter. In our experiments S^ϵ has the size $\sqrt[3]{S^4}$. This limit is a compromise between memory consumption and speed suitable for the test set. The compromise is faster than using the full S as suggested by Allis *et al.* [1]. Using S for the limit slows down RP–PN^2 disproportionally when many threads are used because the PN_1 tree grows faster in RP–PN^2 than in the sequential PN^2. Moreover, the size of PN_2 grows rapidly resulting in slowing down RP–PN^2. An advantage of using the above limit compared to Breuker's method [11] is that the former is robust to varying problem sizes. The values for the parameters of RP–PNS were set to $D = 5$ and $\epsilon = 0.75$ based on trial-and-error.

The experiments were carried out on a Linux server with eight 2.66 GHz Xeon cores and 8 GB of RAM. The program was implemented in C++.

5.2 Results

Two series of experiments were conducted. The first series tests the efficiency of RP–PNS; the second tests the efficiency of RP–PN^2.

For comparing the efficiency of different configurations, we selected a subset of the 143 problems for which PNS was able to find a solution in less than 30 seconds. This selection enabled us to acquire the experimental results for the series of experiments for RP–PN^2 in a reasonable time. We call the set of 143 problems the comparison set, S_{143}. PNS required an average of 4.28 million evaluated nodes for solving a problem of S_{143} with a standard deviation of 2.9 million nodes.

In the first series of experiments we tested the performance of RP–PNS for solving the positions of S_{143}. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the upper part of Table 1. We observe that the scaling factor for 2, 4, and 8 threads is 1.6, 2.5, and 3.5, respectively. Based on the results we compute that the search overhead expressed by the number of nodes evaluated is only ca. 33% for 8 threads. This means that the synchronization overhead is responsible for the largest part of the total overhead. Finally, we see that RP–PNS$_8$ uses 50% more memory than PNS.

In the second series of experiments we tested the performance of RP–PN^2. The results regarding time, nodes evaluated, and nodes in memory for 1, 2, 4, and 8 threads are given in the lower part of Table 1. We observe that the scaling factor for 2, 4, and 8 threads is 1.9, 3.4, and 4.7, respectively. Compared to RP–PNS the relative scaling factor of RP–PN^2 is better for all configurations.

Table 1. Experimental results for RP–PNS and RP–PN^2 on S_{143}. The total time is the time required for solving all problems. "Nodes in memory" is the sum of all M_i, where M_i is the maximum number of nodes in memory used for test problem i. Nodes evaluated is the sum of all nodes evaluated all problems. For RP–PN^2, this includes evaluations in the PN_2 tree and possible double evaluations when trees are re-searched.

	PNS	RP–PNS$_2$	RP–PNS$_4$	RP–PNS$_8$
Total Time (sec.)	1,679	1,072	682	478
Total scaling factor	1	1.6	2.5	3.5
Total nodes evaluated (million)	612	673	745	815
Total nodes in memory (million)	367	423	494	550

	PN^2	RP–PN$^2_2$	RP–PN$^2_4$	RP–PN$^2_8$
Total Time (sec.)	6,735	3,275	1,966	1,419
Total scaling factor PN^2	1	1.9	3.4	4.7
Total scaling factor compared to PNS	0.25	0.52	0.85	1.18
Total nodes evaluated (million)	2,271	2,426	2,534	2,883
Total nodes in memory (million)	68	68	70	73

The search overhead of RP–PN$^2_8$ is 27% which is comparable to the search overhead of RP–PNS$_8$ (33%, cf. above). At the same time the total overhead of RP–PN$^2_8$ is smaller. This means that the synchronization overhead is smaller for RP–PN$^2_8$ than for RP–PNS$_8$. The reason is that more time is spent in the PN_2 trees. Therefore, the probability that two threads simultaneously try to lock the same node of the PN_1 tree is reduced. Finally, we remark that in absolute terms, RP–PN$^2_8$ is slightly faster than PNS.

Despite the fact that RP–PN^2 has a better scaling factor than RP–PNS, RP–PNS is still faster than RP–PN^2 when the same number of threads is used. However, RP–PN^2 consumes less memory than RP–PNS.

5.3 Discussion

It would be interesting to compare the results of the experiments presented in Subsect. 5.2 to the performance of ParaPDS. However, the direct comparison between the results obtained for ParaPDS and RP–PNS is not feasible because of at least three difficulties.

First, the games tested are different (ParaPDS was tested on Othello, whereas RP–PNS is tested on LOA).

Second, the type of hardware is different. As described in Sect. 3, ParaPDS is designed for distributed memory whereas and RP–PNS is designed for shared memory.

Third, ParaPDS is a depth-first search variant of PNS whereas RP–PNS is not. ParaPDS is slowed down because of the transposition tables in distributed memory.

ParaPDS and RP–PN^2 both re-search in order to save memory. When comparing the experimental results for these two algorithms they appear to scale up in the same order of magnitude on a superficial glance. On closer inspection, a direct comparison of the numbers would be unfair. ParaPDS and RP–PN^2 parallelize different sequential algorithms. Furthermore, RP–PN^2 parallelizes transposition tables while our implementation of RP–PNS does not. Moreover, it can be expected that sequential PN^2 profits more from transposition tables than RP–PN^2 because the parallel version would suffer from additional communication and synchronization overhead.

In RP–PN^2 the size of the PN_2 tree determines how much the algorithm trades speed for memory. If the PN_2 is too large, the penalty for searching an unimportant subtree will be too large as well. In our implementation, we chose rather small PN_2 trees because the randomization is imprecise. Moreover, the PN_2 trees are bigger when less threads are used. This explains why RP–PN^2 (with a scaling factor of 4.7) scales better than RP–PNS (with a scaling factor of 3.5). A second factor contributing to the better scaling is the reduced synchronization overhead compared to RP–PNS. This effect is produced by the smaller relative number of waiting threads.

We may speculate that RP–PNS and RP–PN^2 could greatly profit from a more precise criterion for branching from the PV. To that end, it is desirable to find a quick algorithm for finding the k-best nodes in a proof-number tree. Thereby, the true best variations could be investigated.

6 Conclusion and Future Research

In this paper, we introduced a new parallel Proof-Number Search for shared memory, called RP–PNS. The parallelization is achieved by threads that select moves close to the principal variation based on a probability distribution. Furthermore, we adapted RP–PNS for PN^2, resulting in an algorithm we call RP–PN^2.

The scaling factor for RP–PN^2 (4.7) is even better than that of RP–PNS (3.5) but this is mainly because the size of the PN_2 tree depends on the number of threads used. Based on these results we may conclude that RP–PNS and RP–PN^2 are viable for parallelizing PNS and PN^2, respectively. Strong comparative conclusions cannot be made for ParaPDS and RP–PNS.

Future research will address the following four directions. (1) A combined parallelization at PN_1 and PN_2 trees of RP–PN^2 will be tested on a shared-memory system with more cores. (2) A better distribution for guiding the move selection, possibly by including more information in the nodes, will be tested to reduce the search overhead. For instance, the probability of selecting a child N_i could be set to $1-(bsn(N_i)/\sum_{j=1,..,|N|} bsn(N_j))$. (3) The concept of the k-most proving nodes of a proof-number tree and an algorithm for finding these nodes efficiently on a parallelized tree will be investigated. (4) The speedup of reducing the number of node locks by pooling will be investigated.

Acknowledgments. We thank the anonymous referees for their valuable advise. This work is financed by the Dutch Organisation for Scientific Research in the framework of the project Go for Go, grant number 612.066.409.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-Number Search. Artificial Intelligence 66(1), 91–124 (1994)
2. Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands (1994)
3. Nagai, A.: A new depth-first search algorithm for AND/OR trees. In: Matsubara, H., Kotani, Y., Takizawa, T., Yoshikawa, A. (eds.) Proceedings of the Complex Games Lab Workhshop, ETL, Tsuruoka, Japan, pp. 40–45 (1998)
4. Nagai, A.: Df-pn algorithm for searching AND/OR trees and its applications. PhD thesis, University of Tokyo, Japan (2002)
5. Seo, M., Iida, H., Uiterwijk, J.W.H.M.: The PN*-Search algorithm: application to tsume shogi. Artificial Intelligence 129(1-2), 253–277 (2001)
6. Kishimoto, A., Müller, M.: DF-PN in Go: Application to the one-eye problem. In: van den Herik, H.J., Iida, H., Heinz, E. (eds.) Proceedings of the 10th Advances in Computer Games Conference (ACG 2003), Norwell, MA, USA, vol. 10, pp. 125–141. Kluwer Academic, Dordrecht (2003)
7. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science 317(5844), 1518–1522 (2007)
8. Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: PDS-PN: A new proof-number search algorithm: application to Lines of Action. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 170–185. Springer, Heidelberg (2003)
9. Brockington, M.: A taxonomy of parallel game-tree search algorithms. ICCA Journal 19(3), 162–174 (1996)
10. Kishimoto, A., Kotani, Y.: Parallel AND/OR tree search based on proof and disproof numbers. In: Proceedings of the 5th Game Programming Workshop, Hakone, Japan. IPSJ Symposium Series, vol. 99(14), pp. 24–30 (1999)
11. Breuker, D.: Memory versus search. PhD thesis, Maastricht University, The Netherlands (1998)
12. Brockington, M., Schaeffer, J.: APHID Game-Tree Search. In: van den Herik, H., Uiterwijk, J. (eds.) Advances in Computer Chess, Univeriseit Maastricht, vol. 8, pp. 69–92 (1997)
13. McAllester, D.: Conspiracy numbers for min-max search. Artificial Intelligence 35(3), 287–310 (1988)
14. Lorenz, U.: Parallel controlled conspiracy number search. In: Monien, M., Feldmann, R. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 420–430. Springer, Heidelberg (2002)
15. Shoham, Y., Toledo, S.: Parallel randomized best-first minimax search. Artificial Intelligence 137(1-2), 165–196 (2002)
16. Buro, M.: Toward opening book learning. ICCA Journal 22, 98–102 (1999)
17. Chaslot, G., Winands, M.H.M., van den Herik, H.J.: Parallel Monte-Carlo Tree Search. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 60–71. Springer, Heidelberg (2008)
18. Cazenave, T., Jouandeau, N.: On the Parallelization of UCT. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M. (eds.) Computer Games Workshop (CGW 2007). MICC Technical Report Series, vol. 07-06, pp. 93–101. Maastricht University, Maastricht (2007)

19. Pawlewicz, J., Lew, L.: Improving depth-first pn-search: $1+\epsilon$ trick. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 160–170. Springer, Heidelberg (2007)
20. van den Herik, H.J., Winands, M.H.M.: Proof-Number search and its variants. In: Tizhoosh, H., Ventresca, M. (eds.) Oppositional Concepts in Computational Intelligence. Studies in Computational Intelligence, vol. 155, pp. 91–118. Springer, Heidelberg (2008)

Hex, Braids, the Crossing Rule, and XH-Search

Philip Henderson, Broderick Arneson, and Ryan B. Hayward

Dept. of Computing Science, University of Alberta
{ph,broderic,hayward}@cs.ualberta.ca

Abstract. We present XH-search, a Hex connection finding algorithm. XH-search extends Anshelevich's H-search by incorporating a new crossing rule to find braids, connections built from overlapping subconnections.

1 Introduction

Hex is the connection game invented by Piet Hein [1] and John Nash [2]. The board is a rhombus of hexagonal cells. On alternating turns, each player places a stone of her colour on any vacant cell. The winner is the player who connects her two opposing edges with a path of her stones. Hex has many nice properties: additional stones of a player's colour are never disadvantageous, the game cannot end in a draw, and the first player has a winning strategy [2]. However, determining the winner of arbitrary positions is PSPACE-complete [3].

For a Hex position, a *point* is either a vacant cell or a maximal connected set of same-coloured stones; the latter is a *chain*. We assume that board edges are represented by stones, so a chain can include a board edge. In Hex, a common tactical question is whether a specified pair of points can be connected. For a Hex position, a subgame with a second-player strategy (respectively first-player strategy) to connect two points is a *virtual connection* or VC (resp. *virtual semi connection* or SC). For a VC/SC, the two points connected are its *endpoints*, while the set of vacant cells used in the connecting strategy is its *carrier*. For an SC, the initial move of the strategy is its *key*. See Fig. 1.

Anshelevich presented H-search [4,5], a hierarchical VC/SC composition algorithm. H-search is the foundation of Hexy, Six, and Wolve, the only gold medal Hex programs of the Computer Games Olympiad [6,7,8,9,10]. H-search uses three rules, respectively *base/AND/OR*.
(1) Each player has an empty carrier VC between each pair of adjacent points.
(2) If a player has VCs α_1, α_2 with endpoint pairs $\{p_0, p_1\}$,$\{p_0, p_2\}$ and carriers C_1, C_2 such that $C_1 \cup \{p_1\}$ and $C_2 \cup \{p_2\}$ do not intersect, then

(i) if the midpoint p_0 is vacant, then combining α_1, α_2 forms an SC with endpoints p_1, p_2, carrier $C_1 \cup C_2 \cup \{p_0\}$, and key p_0,

(ii) if the midpoint p_0 is a chain for the player, then combining α_1, α_2 forms a VC with endpoints p_1, p_2 and carrier $C_1 \cup C_2$.
(3) If between endpoints p_1, p_2 a player has SCs $\alpha_1, \ldots, \alpha_k$ with carriers $C_1, \ldots, C_k$ such that $C_1 \cap \ldots \cap C_k$ is empty, then combining the SCs forms a VC with endpoints p_1, p_2 and carrier $C_1 \cup \ldots \cup C_k$.

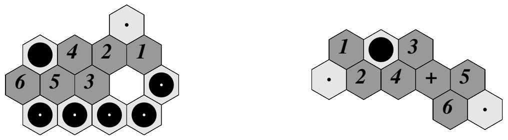


Fig. 1. Diagrams of a Black VC (left) and a Black SC (right). Carriers are shaded, endpoints are dotted, and the SC key is +. In Black's VC strategy, which connects the top dotted cell to the bottom dotted group, if White plays 1 then Black plays 2; then, if White plays 3 then Black plays 4 and then one of {5,6}. In Black's SC strategy, which connects the dotted cells, after playing at the key, Black plays to get one of each of {1,2}, {3,4}, {5,6}.

When used in automated Hex players, H-search is usually restricted by limiting the number of SCs used in the OR rule, for otherwise it takes too long. However, as Anshelevich observed, even unrestricted H-search misses some connections [4,5], including the SC shown in Fig. 2. We call this SC the *braid*, as its substrategies are tangled together. It is of interest to extend H-search in a way that allows for efficient discovery of new connections.

Melis extended Anshelevich's implementation of H-search by allowing board edges as AND rule midpoints [11,9]. Rasmussen et al. extended H-search by triggering a VC search if the OR rule finds an SC set with small carrier intersection [12]; this finds more connections but the search time grows exponentially in the number of cells in the search. Yang described a decomposition notation for Hex connections, including his hand-crafted centre-opening 9×9 SC [13,14], and Noshita introduced union-connections, which are useful in connection verification but not found by H-search [15,16]; neither of these techniques has been used in an automated connection discovery algorithm.

In this paper we present XH-search, an easily implemented extension of H-search. XH-search finds all connections found by H-search, as well as connections built up from braids. The Crossing Rule, a new Hex connection deduction rule, allows for an efficient implementation of XH-search.

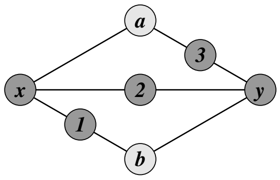


Fig. 2. The braid, an SC not found by H-search. Endpoints are a, b; the key is x or y. To connect, say after playing at key x, if the opponent blocks at 1 then respond at y and then claim one of $\{2, 3\}$. Notice that within the braid there are VCs between each of $\{a, x\}$, $\{b, y\}$, and SCs between each of $\{x, y\}$, $\{x, b\}$, $\{a, y\}$.

2 Stepping Stones

In order to describe the Crossing Rule we first need to define stepping stones. Although defined as points that are internal to a connection, we will use them later as braid endpoints.

Stepping Stone. For a chain X and VC or SC C, X is a *stepping stone* of C if following C's strategy guarantees that C's endpoints will be connected by a chain containing X.

For a VC/SC C, $SS(C)$ denotes the set of C's stepping stones. See Fig. 3.

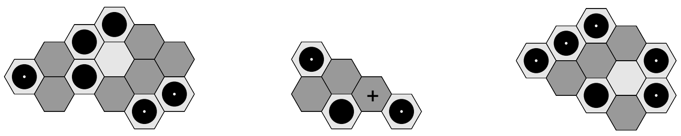


Fig. 3. A VC (left) and SC (middle) with stepping stone. A VC (right) with none.

Lemma 1. *Let C be a base rule VC. Then C has no stepping stone: $SS(C) = \emptyset$.*

Proof. In a base rule VC the two endpoints are neighbours and so already connected. □

Lemma 2. *Let C be a VC computed via the OR rule from SCs $C_1, \ldots, C_k$. Then each chain which is a stepping stone for every C_j is a stepping stone of C: $\cap_{j=1}^{k} SS(C_j) \subseteq SS(C)$.*

Proof. If a chain X is a stepping stone for every SC C_j then, regardless of which SC is maintained, there will be a chain connecting C's endpoints, equal to C_j's endpoints, that contains X. □

Lemma 3. *Let C be an SC with key k computed via the AND rule from VCs C_1, C_2 with vacant midpoint k. Then any chain which is a stepping stone of C_1 or C_2 is a stepping stone of C: $SS(C_1) \cup SS(C_2) \subseteq SS(C)$.*

Proof. The strategy that maintains C first plays at k and then maintains both C_1 and C_2. C_1 connects k to one endpoint, say p_1, of C and C_2 connects k to the other endpoint, say p_2. Each chain created by C also follows C_1 and so connects p_1 to k and contains each stepping stone s_1 of C_1; similarly, it follows C_2 and so connects p_2 to k and contains each stepping stone s_2 of C_2. □

Notice in Lemma 3 that k is vacant, so not in a chain, so not in $SS(C)$.

Lemma 4. *Let C be a VC or SC computed via the AND rule from connections C_1, C_2 with chain midpoint X. Then $SS(C_1) \cup SS(C_2) \cup \{X\} \subseteq SS(C)$.*

Proof. Following the strategies for C_1 and C_2 ensures that the union of X with connecting chains for C_1 and C_2, each of which connects X to an endpoint of C, forms a connecting chain for C containing X. Thus X is in $SS(C)$. The inclusion of $SS(C_1)$ and $SS(C_2)$ in $SS(C)$ follows as in the proof of Lemma 3. □

For each connection C discovered in XH-search we compute a subset $SS^*(C)$ of $SS(C)$. $SS^*(C)$ is defined by applying the preceding lemmas, with this exception: for each VC C computed via the OR rule, $SS^*(C)$ is the empty set. We refer to this as the *SS algorithm.*

Lemma 5. *Let C be a connection with endpoints p_1, p_2. Then for any stepping stone s in $SS^*(C)$ there is a partition S_1, S_2 of the carrier of C such that S_1 is the carrier of a VC from s to p_1, and if C is a VC (resp. SC) then S_2 is the carrier of a VC (resp. SC) from s to p_2.*

Proof. Argue by induction. If C is a base rule VC then $SS^*(C)$ is empty and the lemma holds vacuously. Assume next that C is a VC built from connections of which the stepping stones satisfy the lemma. If C is built by the OR rule, then $SS^*(C)$ is empty and again the lemma holds vacuously. Assume then that C is built by the AND rule, say from VCs C_1, C_2 with midpoint chain p_0 and endpoint pairs $\{p_0, p_1\}, \{p_0, p_2\}$. Then $SS^*(C) = SS^*(C_1) \cup SS^*(C_2) \cup \{p_0\}$. If $s = p_0$, then partitioning C into C_1, C_2 satisfies the lemma. Assume next that s is in $SS^*(C_1)$. Then by the induction hypothesis, the carrier of C_1 can be partitioned into A, B with A a VC from s to p_1 and B a VC from s to p_0. By the AND rule, taking the union of B and C_2 with midpoint p_0 yields a VC from s to p_2 that is disjoint from A, and the lemma holds. Similarly, the same holds if s is in $SS^*(C_2)$. Thus, by induction, the lemma holds for VCs. The proof is similar for SCs. □

It is of interest to know whether Lemma 5 holds if one replaces $SS^*(C)$ with $SS(C)$. This is the case if the OR rule is restricted to combining two SCs.

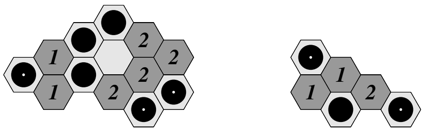


Fig. 4. Illustrating Lemma 5. The VC carrier (left) partitions into S_1, S_2, where each S_j is the carrier of a VC to the stepping stone. The SC carrier (right) partitions into S_1, S_2, where S_1 (resp. S_2) is the carrier of a VC (resp. SC) to the stepping stone.

3 The Crossing Rule

Observe in Fig. 2 that if the endpoints a, b of a braid are same-coloured chains then the braid "untangles" into three disjoint SCs between the internal vacant cells x, y such that two of these SCs have stepping stones. The following rule shows that finding two vacant cells with three such SCs is sufficient to conclude the existence of an SC between particular pairs of the SC's stepping stones.

Crossing Rule. Consider a Hex position with pairwise disjoint SCs C_1, C_2, C_3, each with vacant cell endpoints x, y, and with both $S_1 = SS^*(C_1) \setminus SS^*(C_2)$ and $S_2 = SS^*(C_2) \setminus SS^*(C_1)$ nonempty. Then for any endpoints s_1, s_2 in S_1, S_2 there is an SC C of which the carrier is the union of $\{x, y\}$ with the carriers of C_1, C_2, C_3, and with key x or y.

SCs found by the Crossing Rule are shown in Fig. 5.

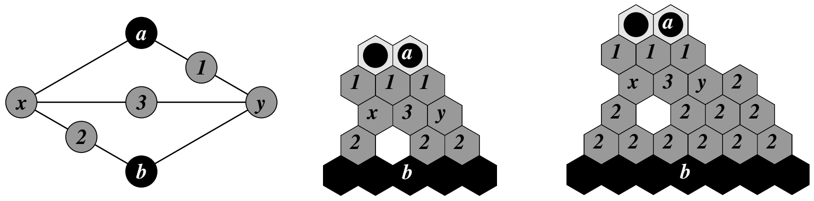


Fig. 5. Crossing Rule SCs. For $j = 1, 2, 3$, cells labelled j form carrier C_j of SC between x, y. $SS^*(C_1)$ contains a and not b. $SS^*(C_2)$ contains b and not a. Combining these SCs yields SC between a, b with carrier $\{x, y\} \cup C_1 \cup C_2 \cup C_3$ and key x or y.

Proof. By Lemma 5 we can partition C_1's carrier into VC V_1 and SC W_1, and partition C_2's carrier into VC V_2 and SC W_2. Assume first that VCs V_1, V_2 have a common endpoint: each connects their respective stepping stone endpoint s_1, s_2 to the same vacant cell, say x. Then by the AND rule, there exists an SC C^- connecting s_1 to s_2 with key x and of which the carrier is the union of $\{x\}$ and the carriers of V_1, V_2. C is the same as C^-, except with an (unnecessarily) larger carrier, so C can follow the same strategy as C^- and we are done.

Assume next that V_1 and V_2 have no common endpoint. By relabelling cells if necessary, assume V_1 connects s_1 to x and V_2 connects s_2 to y. Thus W_1 connects s_1 to y and W_2 connects s_2 to x. Notice that the carriers of V_1, W_1, V_2, W_2, C_3 are pairwise disjoint by construction. The strategy to maintain SC C is as follows: play the key x as the first move, and then maintain VCs V_1, V_2 against any probes into their carriers. If the opponent's first probe outside V_1, V_2 is in W_1 or C_3, then playing the key of W_2 completes the connection from s_1 to x to s_2 via V_1, W_2. If instead the opponent's first probe is in W_2, then play y next, noting that y has a VC to s_1 via application of the OR rule to two disjoint SCs of which the carriers are W_1 and the series combination of C_3, V_1 through x. Since y also has a disjoint VC to s_2 (V_2), we are done. □

Notice that if the second case of the Crossing Rule's proof applies then either x or y can be key. Also, if the first case of the proof applies, then key selection matters but C is a connection of which the carrier properly contains the carrier of a connection that can be deduced via H-search. We assume that H-search is implemented so that a connection is deleted whenever a second connection is discovered with the same endpoints but with a carrier that is a proper subset of the first connection's carrier.[1] Thus the first case is not relevant, and we can

[1] Such connections are provably useless and are pruned by Hex programs such as SIX and WOLVE [11].

assume that any SC discovered by the Crossing Rule can have either x or y as its key. Also, the Crossing Rule deduces an SC joining any such s_1, s_2. Thus we can compute a single carrier and key, and then add the "same" SC to various different pair-connection lists. This also holds if C_3 has stepping stones. The Crossing Rule building block SCs do not share any endpoints with the deduced SC: in some sense, connections deduced by the Crossing Rule are orthogonal to their subconnections, whereas the connections deduced by the AND/OR rules are parallel to their subconnections.

The Crossing Rule can be strengthened. A cell is *dead* if it is not on any minimal path connecting either player's two edges. A set of vacant cells is *captured* by a player if she has a second player strategy on that set that leaves every opponent stone in that set dead. For example, the carrier of an *edge bridge*, shown in Fig 6, is captured. Filling in a player's captured set with her stones does not change the winner of a postion [17,18].

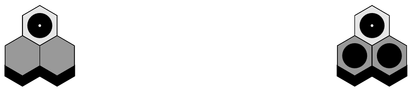


Fig. 6. A black edge bridge (left). Black fill-in does not alter the winner (right).

Strong Crossing Rule. Consider a Hex position with SCs C_1, C_2, C_3, each with vacant cell endpoints x, y, and with both $S_1 = SS^*(C_1) \setminus SS^*(C_2)$ and $S_2 = SS^*(C_2) \setminus SS^*(C_1)$ nonempty. Further, assume that the player with these SCs captures a set B by playing at y, and that $C_1 \cap C_2$, $C_1 \cap C_3$, $C_2 \cap C_3$ are each a subset of B. Then for any endpoints s_1, s_2 in S_1, S_2 there is an SC C of which the carrier is the union of $\{x, y\}$ and B with the carriers of C_1, C_2, C_3, and with key y.

Proof. (sketch) The result follows from the Crossing Rule and the fact that filling in captured sets does not change the winner of a game. □

An SC found by the Strong Crossing Rule is shown in Fig. 7. XH-search applies the Strong Crossing Rule by checking whether either of the potential keys x or y forms a bridge with the edge; the carrier of the edge bridge is the captured set B. When XH-search finds such an SC C, it updates $SS^*(C)$ by applying the next lemma.

Lemma 6. *Let C be an SC computed via the (Strong) Crossing Rule on SCs C_1, C_2, C_3. Then $SS(C)$ is empty.*

Proof. Recalling the proof of the Crossing Rule, there is no common substrategy among the potential outcomes: any portion of C_1, C_2, C_3 could be omitted from a winning path. Indeed, even the intersection of any of these SCs cannot be valid as their strategies were partitioned into disjoint carriers. Thus, in general we cannot conclude the existence of any stepping stones for an SC deduced from the (Strong) Crossing Rule. □

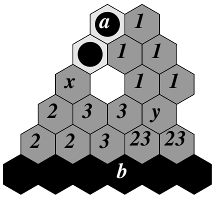


Fig. 7. An SC found by the Strong Crossing Rule. For $j = 1, 2, 3$, cells labelled j form carrier C_j of SC between x, y. Cells labelled both 2,3 form set B and are captured if Black plays y. $SS^*(C_1)$ contains a and not b. $SS^*(C_2)$ contains b and not a. Combining these SCs yields an SC between a, b with carrier $\{x, y\} \cup B \cup C_1 \cup C_2 \cup C_3$ and key y.

Using this lemma in the SS algorithm does not change the validity of Lemma 5, which holds vacuously for any empty set of stepping stones. Thus the (Strong) Crossing Rule still holds.

4 Crossing Rule Connections

We now show some Hex connections found by XH-search but not H-search. Our implementation of XH-search computes stepping stones via the SS algorithm, with each connection storing all of their stepping stones, and applies the Strong Crossing Rule with edge bridges as the only captured sets considered. The three VCs in Fig. 8 are common edge VCs, also known as templates (see King's webpage for more templates [19]). In each of these examples H-search fails to find an SC which does not use the marked cell. These respective three "missing SCs", found by XH-search, are those shown in Figs. 5 and 7.

If the Crossing Rule could only find common edge VCs, then adding a library of VC patterns to check would be an effective alternative. Thus, in Fig. 9 we show more connections from games played by our Hex programs in which XH-search proved advantageous.

The Crossing Rule requires both endpoints of the deduced SC to be chains, and most connections found by XH-search but not H-search are near an edge. Thus, when using XH-search we ensure that the AND Rule allows edges as midpoint; this allows edges to be stepping stones.

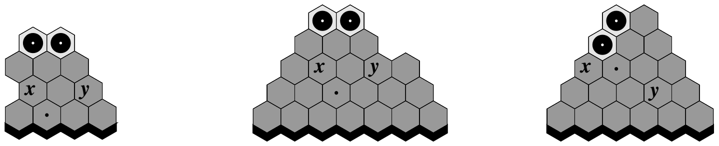
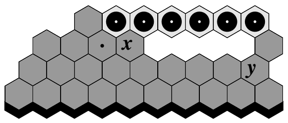


Fig. 8. Edge VCs found by XH-search but not H-search. Each SC found by H-search uses the marked cell. The "missing SCs", which do not use this cell, are in Fig. 5 or 7.



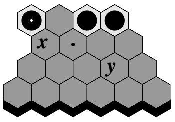


Fig. 9. More edge VCs found by XH-search but not H-search. Each SC found by H-search uses the marked cell. The two "missing SCs" are each similar to that of Fig. 7.

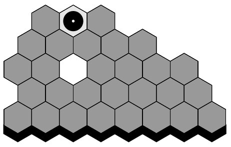
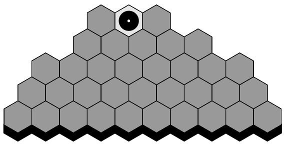



Fig. 10. XH-search finds neither the SC (left) nor the VC (right)

XH-search is not complete: there are connections, some easily recognizable by humans, that it cannot find. See Fig. 10. It is of interest to find some efficient connection recognition algorithm which can find all connections recognizable by human players.

5 Implementation and Complexity

It is straightforward to implement XH-search by starting with H-search, adding stepping stone deductions to the AND/OR rules, and adding the (Strong) Crossing Rule. In the following pseudocode, the queue holds endpoint pairs, each of which has a VC carrier list and an SC carrier list; omitting the crossing rule computation leaves H-search.

```
Algorithm XH-search.
   initialize VC/SC carrier lists:
       for each pair E of endpoints
           E.VCList.makeEmpty(); E.SCList.makeEmpty();
   initialize queue Q with base VCs:
       Q.makeEmpty()
       for each pair E of adjacent endpoints
            Q.add(E); E.VCList.add(baseVC(E))
   while (not Q.isEmpty())
       E <- Q.remove()
       compute crossing rule on E's SCs:
           for each new SC Z with endpoint pair F found,
              Q.add(F); F.SCList.add(Z)
       compute OR rule on E's SCs:
           for each new VC Z found
```

```
            E.VCList.add(Z)
        compute AND rule on E's VCs with both of E's endpoints:
            for each new VC/SC Z with endpoint pair G found,
                Q.add(G); G.(VC/SC)List.add(Z)
    end while
end XH-search
```

Extending H-search by adding stepping stone deductions does not increase the runtime complexity.

Lemma 7. *Let $f(n)$ be the worst-case running time of H-search on a board with n cells. Then the worst-case running time of H-search with stepping stone deductions is in $O(f(n))$.* □

Regarding memory requirements, our H-search implementation stores the two endpoints and carrier for each connection. In order to store stepping stones, two options are possible: store all stepping stones in the same manner as the carrier, or store only a single stepping stone per connection. The second solution is appealing, since a complete set of stepping stones is not required in order to find new connections, and since many connections have few stepping stones; however, it forces us to choose which stepping stone to keep if there are several available. The first solution provides more information, but nearly doubles the memory per connection; the second increases the memory per connection only marginally. Since memory is not a bottleneck for our Hex program, we opted for the first solution.

The running time of XH-search depends not only on the number of points and the sizes of connection lists, but also on the number of discovered connections as well as their type (VC or SC). Nonetheless, we can at least analyze the relative computational efficiency of the different deduction rules in terms of the known factors.

Aside from the AND rule, OR rule, and Crossing Rule, we also include naive deduction of missing SCs of the form shown in Fig. 2. This naive deduction would involve finding four distinct points a, b, x, y such that x, y are vacant, and such that there exists five pairwise disjoint connections with two VCs joining pairs a, x and b, y and three SCs joining pairs a, y and b, x and x, y. Since the OR rule's complexity is parameterized by the maximum number of SCs it may combine in parallel, we have included entries for parameter values of both three and four, which are the common selections for Hex programs. Table 1 summarizes our analysis, with n representing the number of points and l_V, l_S representing the list sizes for VCs and SCs, respectively. We distinguish between these two list lengths, as the number of SCs usually far exceeds the number of VCs.

We can now clearly see the benefit of using stepping stones: the complexity of the Crossing Rule is roughly the square root of a naive implementation, and roughly the same order of complexity as the regular deduction rules in H-search.

Table 1. Worst-case runtime (within constant factor) of connection deducing rules

Deduction rule	running time
AND rule	$n^3 l_V^2$
OR-3 rule	$n^2 l_S^3$
OR-4 rule	$n^2 l_S^4$
crossing rule	$n^2 l_S^3$
naive rule	$n^4 l_V^2 l_S^3$

6 Experiments

To test the effectiveness of the crossing rule, we played an 11×11 tournament between Monte Carlo players H (using H-search) and XH (using XH-search). Each player is a version of MoHex, the UCT Hex player that won silver at the 2008 Computer Games Olympiad in Beijing [7]. Each player uses 100K rollouts to analyze UCT tree nodes and computes connections with our normal tournament settings: the AND rule is computed over the edge (so the edge of the board can be the midpoint of an AND connection) and the OR-rule combines up to 4 SCs. Neither player uses an opening book or endgame solver. The tournament comprises 4-rounds, with each player opening at each position once as Black and once as White (with no swap move allowed), for a total of 4×121×2=968 games.

XH defeated H 501 games to 467, 17 games above a breakeven score of 484, taking on average 1.099 times as long as H per move. The time increase is roughly in line with the analysis in Table 1: an 11 × 11 board has roughly $n = 100, l_V = 10, l_S = 25$, and finding more connections with the Crossing Rule increases the number of iterations for all rules.

The crossing rule thus added roughly 12 ELO points in strength in exchange for 9.9% more computation time on average. While this gain might seem negligible, strength gains measured via all-opening no-swap tournaments are muted due to the forced inclusion of many very strong and very weak opening moves. By comparison, each doubling of MoHex's number of rollouts results in an average ELO gain of 31.8 (averaged over 7 rollout doublings, from 1s to 128s, when competing against a 1s version). Thus, per unit time invested, the crossing rule represents a more significant strength gain than simply increasing the number of rollouts.

7 Conclusions

XH-search is efficient and easily implemented. Furthermore, it identifies important Hex connections that cannot be found with H-search.

In future work we hope to identify further efficient deduction rules, particularly those that identify the most common connection omissions. The captured sets of the Strong Crossing Rule need not be restricted to edge bridges; it would be interesting to find other efficient methods that allow for carrier overlap within connection deduction rules.

Acknowledgements

We gratefully acknowledge NSERC and iCORE for funding this work, and the University of Alberta's Hex and GAMES group members for providing support and feedback.

References

1. Hein, P.: Vil de laere Polygon? Series of articles in *Politiken* newspaper (December 1942)
2. Nash, J.: Some games and machines for playing them. Technical Report D-1164, RAND (1952)
3. Reisch, S.: Hex ist PSPACE-vollständig. Acta Informatica 15, 167–191 (1981)
4. Anshelevich, V.V.: The game of Hex: An automatic theorem proving approach to game programming. In: AAAI/IAAI, pp. 189–194 (2000)
5. Anshelevich, V.V.: A hierarchical approach to computer Hex. Artificial Intelligence 134(1–2), 101–120 (2002)
6. Anshelevich, V.V.: Hexy wins Hex tournament. ICGA Journal 23(3), 181–184 (2000)
7. Arneson, B., Henderson, P., Hayward, R.B.: Wolve Wins Hex Tournament. ICGA 31(4) (2008)
8. Hayward, R.B.: Six wins Hex tournament. ICGA Journal 29(3), 163–165 (2006)
9. Melis, G., Hayward, R.: Six wins Hex tournament. ICGA Journal 26(4), 277–280 (2003)
10. Willemson, J., Björnsson, Y.: Six wins Hex tournament. ICGA Journal 27(3), 180 (2004)
11. Melis, G.: Six (2006), http://six.retes.hu/
12. Rasmussen, R., Maire, F.: An extension of the H-search algorithm for artificial Hex players. In: Australian Conference on Artificial Intelligence, pp. 646–657 (2004)
13. Yang, J.: Jing Yang's web site (2003), http://www.ee.umanitoba.ca/~jingyang/
14. Yang, J., Liao, S., Pawlak, M.: A decomposition method for finding solution in game Hex 7x7. In: ADCOG, pp. 96–111 (2001)
15. Kohei, N.: Union-connections and a simple readable winning way in 7×7 Hex. In: Proceedings of 11th Game Programming Workshop, pp. 72–79 (2004)
16. Kohei, N.: Union-connections and straightforward winning strategies in Hex. ICGA Journal 28(1), 3–12 (2005)
17. Hayward, R.B.: A note on domination in Hex. Technical report, University of Alberta (2003)
18. Hayward, R.B., Björnsson, Y., Johanson, M., Kan, M., Po, N., van Rijswijck, J.: Solving 7×7 Hex: Virtual connections and game-state reduction. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games. International Federation for Information Processing, vol. 263, pp. 261–278. Kluwer Academic Publishers, Boston (2003)
19. King, D.: Hall of hexagons - the game of Hex (2007), http://www.drking.plus.com/hexagons/hex/index.html

Performance and Prediction: Bayesian Modelling of Fallible Choice in Chess

Guy Haworth[1], Ken Regan[2], and Giuseppe Di Fatta[1]

[1] School of Systems Eng., Univ. of Reading, RG6 6AH, UK
`{g.haworth,g.difatta}@reading.ac.uk`
[2] Dept. of CS and Eng., Univ. at Buffalo, State Univ. of New York, Buffalo, NY
`regan@cse.buffalo.edu`

Abstract. Evaluating agents in decision-making applications requires assessing their skill and predicting their behaviour. Both are well developed in Poker-like situations, but less so in more complex game and model domains. This paper addresses both tasks by using Bayesian inference in a benchmark space of reference agents. The concepts are explained and demonstrated using the game of chess but the model applies generically to any domain with quantifiable options and fallible choice. Demonstration applications address questions frequently asked by the chess community regarding the stability of the rating scale, the comparison of players of different eras and/or leagues, and controversial incidents possibly involving fraud. The last include alleged under-performance, fabrication of tournament results, and clandestine use of computer advice during competition. Beyond the model world of games, the aim is to improve fallible human performance in complex, high-value tasks.

1 Introduction

In the evolving world today, decision-making is becoming ever more difficult. Professionals are increasingly working as parts of man-machine systems, helped or supplanted by intelligent, carbon agents. Those responsible for the quality of the decisions therefore have a need to (a) assess the quality of their agents, and (b) predict the probabilities of other agents' choices in 'zero sum' situations. These needs are clear in real-time financial scenarios – city markets, auctions, casinos - and for effective control of utility services.

A method is proposed here for modelling and analysing decision-making in complex but quantifiable domains. The 'model world' of chess serves, as it has often done in the past, as a demonstration domain.

Skill in the global chess community has been measured by the FIDE Elo system [1] on the basis of past results. However, a good player needs to assess their opponents' skill of the moment, and chooses a move which is worst for the opponent rather than best in an absolute chessic sense. The human factor is perhaps more evident in a game of Poker or Roshambo.[1] Skill assessment is an analysis of the past, but performance prediction more dynamically considers the parameters of the current situation. One

[1] Roshambo is also known as *Rock, Paper, Scissors*, a pure exercise in opponent assessment.

might consider that the better choices are more likely than worse ones, but that the less the apparent skill or rationality of the decision-maker, the more likely the worse choices are to be made.

The proposed modelling method uses a *Benchmark Space* and a *Bayesian Inference* mapping of behaviour into that space. The *space* is seeded by Reference Agents which have or are given defined dimensions of fallibility. Bayes' method is used to profile the decision-maker in terms of fallible agents. Thus, the decision-maker or agent analysed is not only positioned relative to other agents and possible threshold performance levels but is also rated in an absolute sense. The demonstration applications in the chess domain address frequently asked questions and some topical issues concerning various forms of cheating. One of these, somewhat ironically, is the illicit use of computer advice during competition.

The Bayesian approach was first proposed [2] in the subdomain of chess where perfect information about the quality of the moves is known. Extending it to chess generally [3, 4, 5] requires resort to fallible benchmarks yielding confident rather than certain results. Nevertheless, [5] shows strong correlation between the current FIDE Elo rating scale and the new *apparent competence* rating *c*.

Section 2 defines the two concepts of Agent Space and Bayesian Inference Mapping, and notes a missed opponent-modelling opportunity. Section 3 extends the principle to that part of chess where engines evaluate positions heuristically. Section 4 reviews the application of the theory in the laboratory and to chess questions of interest. In summarizing, we anticipate the evolution of the approach, its further application in chess, and its use in non-game 'real world' scenarios.

2 Absolute Skill in the Chess Endgame

The Chess Endgame is defined here as that part of chess for which Endgame Tables (EGTs) have been computed. An EGT gives the theoretical value and Depth to Goal of every legal position for an endgame force, e.g., King & Queen v King & Rook (KQKR). The most compact and prevalent EGTs are those of Nalimov [6], providing Depth to Mate (DTM) where mate is the end-goal of chess: these are used by many chess engines on a simple look-up basis. EGTs for all required 3-, 4-, 5- and 6-man endgames are available up to KPPKPP.

Given this database of perfect information, some questions suggest themselves: (a) how difficult are various endgames, (b) how long might a hypothetical fallible agent take to win a game, and (c) how well do humans play endgames?

Although Jansen [7] had addressed the topic of Opponent Fallibility, it was left to Haworth [2] to define an agent space SRFEP of Reference Fallible Endgame Players (RFEPs) as defined in the next section.

2.1 The Agent Space SRFEP of Reference Fallible Endgame Players

Let E be an engine playing an endgame using an EGT: further, let E have a theoretical win of depth d_0. Let $E(c)$ be a stochastic variant of E with apparent competence c, constrained to retain the win[2] but choosing its moves by the following algorithm:

[2] The extension of the model to drawing/losing moves has been done but is not needed here.

- let $\{m_j\}$ be the available winning moves, respectively, to depths $\{d_j\}$,
- move-indexing: $i < j \Rightarrow d_i \leq d_j$, i.e., lower-indexed moves are 'no worse',
- let Prob[$E(c)$ chooses move m_j] $\propto$ Likelihood[m_j] $\equiv L(j, c) \equiv (1 + d_j)^{-c}$.

The space SRFEP of RFEPs satisfies the following requirements:

a. centred: $E(0)$ is a zero-skill agent – all moves are equally likely,
b. ordered: $c_1 < c_2 \Rightarrow$ E[d | $E(c_1)$ moves] $\geq$ E[d | $E(c_1)$ moves]
c. complete: $E(\infty)$ infallibly chooses a best move: $E(-\infty)$ is anti-infallible,
d. sensitive: if $d_{j+1} = d_j+1$, as $d_j \to \infty$, $L(j, c)/L(j+1, c) \to 1$ downwards, and
e. non-exclusive: all moves have a non-zero probability of being chosen.[3]

Three factors make SRFEP 1-dimensional, simplifying its use. Because chess engines consult the EGT directly, their specific search heuristics, search depths, and evaluations are neither relevant, nor is there a perceived need to generalize to $(\kappa + d_j)^{-c}$ with $\kappa > 0$.

Haworth [2] also modelled the endgame as a Markov Space and move-choice as a Markov Process to answer questions 'a' and 'b' above and to show where the more difficult depths of an endgame were.

2.2 Mapping a Player to the Agent Space SRFEP

Question 'c' was answered by rating a player *PL's* play on the basis of an observed set of moves $M \equiv \{M_i\}$. This was done by mapping *PL*, given *M*, to a profile of engines $\{E(c)\}$ in SFREP.

Let us assume that the moves M_i, in fact played by *PL* in the endgame on whatever basis, have in fact been played by an engine $E \equiv E(c)$ where c is one of $\{c_k\}$, e.g., c=0, …, 50. Let the initial probability that $E \equiv E(c_k)$ be $p_{k,0}$: for example, the 'know nothing' stance would set all $p_{k,0}$ to the same value.

Note now that, given that move M_1 is chosen from the moves m_{1j} available:

- Prob[$E \equiv E(c_k)$] $= p_{k,l-1}$ before move M_l is chosen on the lth turn,
- $q_k \equiv$ Prob[$E(c_k)$ chooses move M_l] may be calculated as follows …
- $q_k \equiv (1 + d_l)^{-c} / \Sigma_j (1 + d_j)^{-c}$ where j ranges over the move-options available,
- Bayes' Rule defines the *a posteriori* probability $p_{k,1}$ that $E \equiv E(c_k)$ given M_l,
- As k varies across the range of engines E_α, $p_{k,l} \propto (p_{k,l-1} \times q_k)$,
- $p_{k,l} \equiv (p_{k,l-1} \times q_k) / \Sigma_\alpha(p_{\alpha,l-1} \times q_\alpha)$ where α ranges over the possible engines E_α,
- after observing all moves M_i, Prob[$E \equiv E(c_k)$] $\equiv p_{k,n} \equiv r_k$.

On the evidence of $M \equiv \{M_i\}$, player *PL* has been profiled in the agent space: it has been associated by a player-agent mapping *PA,* with $\{r_k E(c_k)\}$, a probability distribution of agents. In fact, engine *PA*(*PL*) may be defined as $\{r_k E(c_k)\}$, an engine which behaves like engine $E(c_k)$ on each move with probability r_k. We also have a metric for the absolute competence of *PL* in the competence rating $r_{PL} \equiv \Sigma r_k \times c_k$.

Given a fallible opponent *PL*, a player, especially if a computer engine, may model *PL*, predict their behaviour and exploit their apparent weaknesses accordingly [7, 8].

[3] The '1' in $(1 + d_j)$ ensures a non-zero denominator when $d_1 = 0$.

2.3 Adapting to the Opponent: A Missed Opportunity

In 1978, Ken Thompson armed his chess engine BELLE with a secret weapon, the KQKR EGT[4]. The KQ-side has a tough challenge [9, 10] with a budget of 50 moves to capture or mate, and 31 being needed in the worst case. Thompson wagered $100 that no-one would beat BELLE in the KQKR endgame [11-14] and only GM Walter Browne took up the two-game test. Browne failed in the first game but, rising to the competitive challenge, and partially informed by BELLE's KQKR-listings and a plan, he returned to recapture the Rook and his $100 just in time on the 50th move.

Haworth [2] gives details of the moves and progress in depth terms, and analyses them as above, as if the choices of some engine in the set $\{E(0), E(1), \ldots, E(50)\}$[5]. Had BELLE perceived Browne's *apparent competence* c_{WB}, it could have chosen correctly between DTC-optimal moves four times. In fact, it picked the right move just once, missing three opportunities to prolong its defence by the necessary one move.

3 Absolute Skill in Chess

Here are some categories of question that have been asked of human play.

a. Does 'Elo *E*' mean the same today as it did in years past?
b. How does player *PL*'s absolute skill vary over their career?
c. How does player *PL*'s skill compare with others' skill?
d. How do the games of tournament *T* compare with each other?
e. Is player *PL* demonstrating 'Fidelity to a Computer Agent' [15] …
   … in the context of *PL*'s (suspected) clandestine behaviour?

The answers are necessarily statistical and therefore their expected accuracy and the confidence that can be placed in them depends on the amount of data available[6] and its use. Game results and rating changes say little and conflate the behaviour of the two players. The many move-decisions potentially enable a better assessment of player performance in the context of consistent chess engine analysis.

The core idea in [3, 4] is to use chess engines as benchmark agents, assessing human competence on the evidence of their move-decisions and in the context of the engines' assessment of the options. The engines can rarely see a 'win in *n* moves' as in the endgame, and therefore indicate advantage and the consequent likelihood of a win, draw or loss in units of a Pawn. Note three complicating factors in comparison with the endgame-play rating challenge just discussed:

1. the engines' heuristic position evaluations vary from engine to engine,
2. for one engine, the evaluations usually vary with depth of search, and
3. deeper evaluations are more accurate but none are definitive.[7]

[4] In fact, computed to *Depth to Conversion* (DTC), i.e. *depth to capture and/or mate*.
[5] The *c*-bounds 0 and 50, and the Δc choice of 1 are such as not to over-influence the results.
[6] A result *ac* times more accurate is expected to require ac^2 more input data.
[7] Evaluations are merely substitutes for unattainable, perfect win/draw/loss information.

These specific questions indicate the range of questions now being addressed:

Re ‘a’: competence of 1971-1981 ‘Elo 2400 players’ v those of 1996-2006?
Re ‘b’: what is the profile of Victor Korchnoi’s skill over the years?

How do the best performances of World Chess Champions compare?
Guid and Bratko [16] address this using only apparent ‘move error’ as a metric.

Re ‘c’: how do the 1948 World Championship games compare with each other?
Re ‘d’: how did the players perform in a tournament: how do the games compare?
Re ‘e’: can we focus on and analyse *suspect play* at the time or later?

The next two sections are analogous to sections 2.1 and 2.2: they define a space of fallible agents and the way player *PL* is associated by a mapping *PA* with an agent profile *E* of engines in an agent space.

3.1 The Agent Space SRFP of Reference Fallible Players

Chess engines search to increasing depths rather than looking up EGTs, and vary in the heuristic position-evaluations they return, the agent space SRFP has in principle two dimensions which SRFEP does not:

1. (discrete) *search-depth*: evaluations at search-depths d_{min}, … , d_{max}, and
2. (discrete) *engine*: engines E_1, … , E_n may ‘seed’ the space SRFP.[8]

As benchmarks preferably demonstrate high-quality behaviour, these engines should have as high an Elo as possible in the various rating schemes for chess engines. The first computations reported here use SHREDDER 10 and TOGA II v1.3.1 to a modest search depth of 10, although Regan [17] reports that TOGA II v.1.3.1 searching to depth 10 won a match[9] against CRAFTY 20.14 searching to depth 12. SHREDDER [18] is a multiple World Computer Chess Champion.

As better engines become available, one would expect the benchmark set of engines to change. For example, FRITZ 5.32 was state-of-the-art circa 1998 [19], but today one would prefer, e.g., RYBKA 3 and SHREDDER 11. For architectural (WINDOWS/LINUX and UCI[10]) and comparability reasons, the computations reported here continue with the original choices of SHREDDER 10 and TOGA II v1.3.1.

For the chess endgame, the non-negative destination depths were converted easily into positive likelihoods: the depths simply became positive denominators in the likelihood function *L*: the greater the depth, the less attractive that option for the winner. Here, position evaluations may be greater, equal to or less than zero: it seems natural to convert these first into positive numbers analogous to *depths* in the endgame. Again, the least attractive, i.e., smallest, evaluations should associate with the largest positive numbers. Thus, with $w = C(v) > 0$ being a conversion function, let

$$j1 < j2 \Rightarrow \text{move } m_{j1} \text{ ‘is’ no worse than } m_{j2} \Rightarrow v_{j1} \geq v_{j2} \Rightarrow w_{j1} \equiv C(v_{j1}) \leq w_{j2} \equiv C(v_{j2}).$$

[8] To date, these dimensions are ‘null’ in our computations: $n = 1$ and $d_{min} = d_{max} = 10$.
[9] 12.5/20 ⇒ TOGA II v1.3.1 (depth 10) is ~90 ELO better than CRAFTY 20.14 (depth 12).
[10] UCI is the Universal Chess Interface [20].

Note that function $C(v)$ potentially involves further parameters, each a dimension of the space SRFP. A caveat is also appropriate here. It is clear that some functions $C(v)$ have properties which are unrealistic in chess terms. For example, Haworth [4] suggested the function $C_1(v_j) \equiv 1 + |v_1| + |v_1 - v_j|$ but when coupled with the likelihood function $L(j, c) \equiv w_j^{-c}$ as in Section 2.1, the following unrealistic situation arises.

- e moves $m_{1/2}$ are to positions with values $v > 0$ and $v_2 = -1$,
- $w_1 = 1 + v$ & $w_2 = 2 + 2v \Rightarrow L(1, c) = (1 + v)^{-c}$ & $L(2, c) = 2^{-c} \times L(1, c)$
- $\therefore \forall v$, Prob[Engine $E(c)$ chooses the better move m_1] $\equiv 1/(1 + 2^{-c})$,
- but in practice, the greater v, the more likely m_1 is to be chosen.

Therefore, Di Fatta et al. [5] used a different $C(v)$:

- $C_2(v_j) \equiv w_j \equiv \kappa + |v_1 - v_j|$ with $\kappa > 0$, with $L(j, c) = w_j^{-c}$ for $E(c)$ as before,
- parameter κ; one more SRFP dimension, has so far been set to 0.1,
- n.b. $L(j, c)$ depends on v_1-v_j but not on v_1 or other v_i, but
- sensitivity requirement 'd' (in §2.1) suggests v_1 as a parameter of L,
- a correlation of (various Elo) players' apparent errors with v_1 is in plan.

The need to create functions $C(v)$ and $L(w)$ introduces the question of what $C(v)$ and $L(w)$ create the best agent-space SFRP, the one which most faithfully models the behaviour modelled. This question is considered further in the next section defining the association of player *PL* with a compound agent in SRFP. To summarise, SRFP is a space of agents or chess engines $E_i(d, \underline{\mathbf{c}})$ searching to depth d and 'dumbed down' by at least one parameter c.

3.2 Mapping a Player to the Agent Space SRFP

The following notation is useful for this section:

- player *PL's* moves $M \equiv \{M_i\}$ from positions $\{P_i\}$ are available for analysis,
- from position P_i, moves m_{ij} to positions P_{ij} are to be considered,[11]
- engine $E_k(d)$ evaluates position P_{ij} as having value v_{ijk} at search-depth d,[12]
- $C(v)$ maps positions values of any value to R^+: $v_1 > v_2 \Leftrightarrow w_1 < w_2$, and
- engine $E(d, \underline{\mathbf{c}})$ plays move m_{ij} with probability $\propto$ likelihood $L(w_{ij}, \underline{\mathbf{c}})$.

Let the hypothesis $H_{kd\underline{c}}$ be that *PL's* moves are played by some engine $E_k(d, \underline{\mathbf{c}})$ which is in a 'candidate engine' subspace *CS* of SRFP. *Prior probabilities* $p_{kd\underline{c}}$ are assigned to the $H_{kd\underline{c}}$ before any moves are analysed. For example, $p_{kd\underline{c}} = constant$ would represent the often adopted 'know nothing' initial stance but different profiles of *priors* may be used to see what the initial beliefs' long-term influences are.

Bayes' Rule is used to calculate what the *posterior probabilities* $p_{kd\underline{c}}$ are (of $H_{kd\underline{c}}$ being true) after observing one or more moves M_i. Let these posterior probabilities be $q_{kd\underline{c}}$.[13] The Bayes Rule of Inference is simply stated:

[11] All legal moves are considered but engines only evaluate the best *MultiPV* moves precisely.

[12] To simplify the notation, some suffices will be suppressed on occasion as 'understood'.

[13] Bayes' contribution was a belief-modifying formula, obviating the need for heuristics.

$E_k(d, \underline{\mathbf{c}}) \in CS$, $Freq_{kd\underline{c}} \equiv Prior\,Prob[H_{kd\underline{c}}$ is true$] \times \text{Prob}[M_i \mid H_{kd\underline{c}}$ is true$]$,

$\text{Prob}[M_i \mid H_{kd\underline{c}}$ is true$] \propto \text{Likelihood}[E_k(d, \underline{\mathbf{c}})$ plays $M_i]$; $SumFreq = \Sigma_{CS}\, Freq_{kd\underline{c}}$,

$Posterior\ Prob[E_k(d, \underline{\mathbf{c}}) \mid M_i$ is played$] \equiv Freq_{kd\underline{c}} / SumFreq$

Thus after modifying the initial $p_{kd\underline{c}}$ to the final posterior probabilities $q_{kd\underline{c}}$, Bayes' Rule has identified a composite agent or engine $E \equiv \langle q_{kd\underline{c}}\, E_k(d, \underline{\mathbf{c}})\rangle$ which, by definition, decides at each move to play with probability $q_{kd\underline{c}}$ as engine $E_k(d, \underline{\mathbf{c}})$. Thus, again, we have a mapping $PA : Player \rightarrow Agent$ associating players, carbon or silicon, with a profile of engines in the agent space SRFA.

If $s_k \equiv \Sigma_{d\underline{c}}\, wd_d \times q_{kd\underline{c}}$[14] and $r_{PL} \equiv \Sigma_k\, we_k \times s_k$, with some engine's perspectives perhaps more weighted than others but with $\Sigma_k\, we_k \equiv 1$, r_{PL} is an absolute rating for *PL* in the context of the benchmark used. It can therefore be used to compare players, carbon and silicon, of different playing leagues and different eras.

However, the competence of *PL* and *PA*(*PL*) are not the same. Errors made by the benchmark engines when in fact *PL* makes the correct decision are seen by the engines as errors made by *PL*, so *PA*(*PL*) will be somewhat less competent than PA. This complicates the otherwise trivial matter of putting humans and chess-engines on the same scale using games that have already been played[15] but Haworth [4] proposes a 'DGPS' approach, reducing error by identifying errors at reference points, to removing most of the error contributed by the inevitably fallible benchmark engines.[16]

4 SRFA: Computations and Applications

The first 'SRFA' production computations inferred the *apparent competence c* of seven *Virtual Elo-e players*[17] [5]: the results show a correlation between c and FIDE Elos, and provide a context in which other inferred *c* may be assessed. Table 1 summarises the input data, the results and the standard deviation of the results which as expected is approximately inversely proportional to the square-root of the amount of input data.

Table 1. The apparent competence *c*, mean and stdev, with details of contributing data

#	Player	Elo_{min}	Elo_{max}	Period	Games	Pos.	c_{min}	c_{max}	μ_c	σ_c	$\sigma_c * Pos^{½}$
1	Elo_2100	2090	2110	1994-1998	217	12,751	1.04	1.10	1.0660	.00997	1.126
2	Elo_2200	2190	2210	1971-1998	569	29,611	1.11	1.15	1.1285	.00678	1.167
3	Elo_2300	2290	2310	1971-2005	568	30,070	1.14	1.18	1.1605	.00694	1.203
4	Elo_2400	2390	2410	1971-2006	603	31,077	1.21	1.25	1.2277	.00711	1.253
5	Elo_2500	2490	2510	1995-2006	636	30,168	1.25	1.29	1.2722	.00747	1.297
6	Elo_2600	2590	2610	1995-2006	615	30,084	1.27	1.33	1.2971	.00770	1.336
7	Elo_2700	2690	2710	1991-2006	225	13,796	1.29	1.35	1.3233	.01142	1.341

[14] The wd_d emphasise an engine's more accurate evaluations at deeper depths: $\Sigma_d\, wd_d \equiv 1$.

[15] '*PL* & *PA*(*PL*)=*E*(*c*) are Elo 2600' & 'Match *E/E*(*c*) $\Rightarrow$ *E* 400 Elo better' $\Rightarrow$ *E* has Elo 3000.

[16] Consider engine *F*, let *PA(F)* = *E*(*c*), and let there be engine matches *E-E*(*c*) and *E-F*. The match results will show the Elo difference between *E, E*(*c*) and *F*.

[17] The *Virtual Elo-e Player* is a composite of many actual Elo *e* (*e* = 2100 ± 10 etc) players.

The SRFA-computation programme is a continuing experiment: the next section is a description of how that experiment has been created and is being managed.

4.1 The Computational Regime

The aims of the computation are to:

- acquire sound input data, and manage it assuredly, correctly and efficiently,[18]
- ensure that experimental results could be conveniently reproduced,
- exploit multiple computer platforms, separating job creation and commissioning,
- ensure that the engines adopted were of as high a quality as possible.

Some examples of chess-specific issues that needed to be managed:

- human players, with a win in hand, play safely rather than optimally:
  - Guid & Bratko [16] reasonably suggest ignoring positions outside [-2, 2],
- the robustness of statistical results from fallible benchmarks must be tested:
  - there was much criticism of [16] on these grounds, but
  - Guid et al. [21] was only a partially successful response to this criticism.

Some examples of Bayesian Inference issues to be managed:

- probabilities need to be held in log-form to postpone underflow,
- setting priors must be consistent if moves/games are to be compared,
- care is required in setting/adapting the range/granularity of the hypotheses …
- otherwise, the prior probabilities will overly affect the posterior probabilities.

4.2 Applications of 'SRFA' Computation

4.2.1 Recognised Human Achievement

Procrustes allows room here for only a sample of the insights which are now possible. Benchmarks based on reference engines enable comparison of play and players of different eras. The Elo scale is thought to have inflated [22] and a comparison of Elo 2400 play in the periods 1971-1981 and 1996-2006 is in hand. The achievements of top players can be profiled, even before the adoption of the Elo scale in 1970: Korchnoi's *c* and Elo are shown[19] in Fig. 1. A comparison of World Champions is possible [16] though, given the quality of top-level play, the plan here is to reduce *benchmark error* and base any analysis on search-depths much greater than 10.[20]

Keres' 0-4 World Championship performance against Botvinnik in 1948 has long been a matter of speculation, as it is rumoured that he was under pressure not to impede the latter's progress to the title. Keres' and opponents' *c* per game have been computed for the 20 games in which he was involved, see Fig. 2.

[18] Over 200,000 positions, their analyses and Bayesian inferences, are held in a datastore.
[19] The trace of Korchnoi's *c* is a running average *c* based on the last 100 games.
[20] Although fallible benchmarks give results with calculable confidence levels [4].

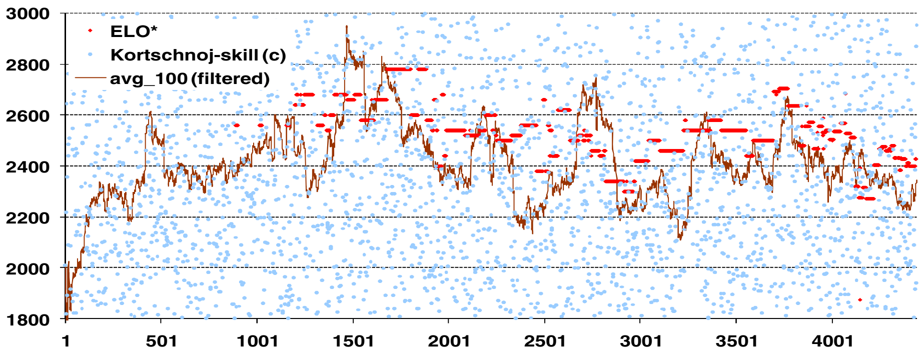


Fig. 1. Korchnoi (1950-): 'FIDE Elo bars' and *apparent competence c* over last 100 games

Questions are asked not only about chess' finest but about the best tournaments, matches, individual performances and games on record. We look forward to identifying games where both sides played conspicuously well whatever the result.

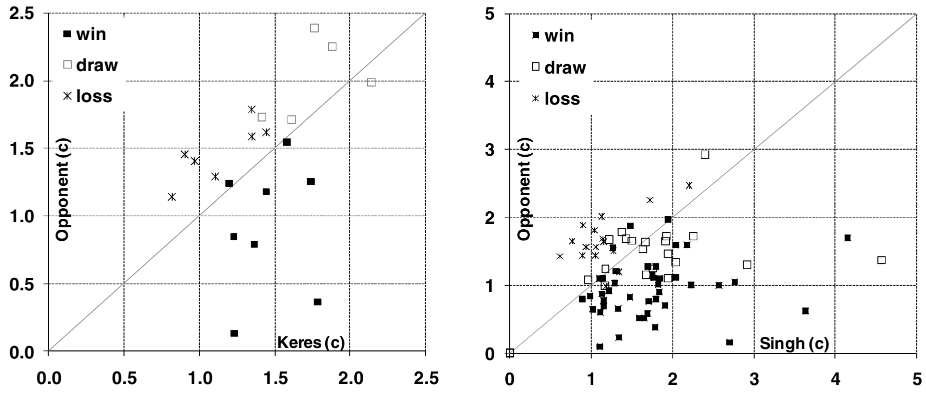


Fig. 2. Left: Keres at the WCC (1948). Right: Some 78 games by D.P.Singh (2005-8).

4.2.2 Alleged Chess Cyborgs

Players suspected of receiving computer advice during play include the following: Clemens Allwermann in 1998 [19], Diwakar Prasad Singh in 2005-6 [23, 24], Eugene Varshavsky at the World Open[21] in 2006 [25], and Krzysztof Ejsmont in 2007 [26]. In all cases, no physical evidence was found[22], the circumstantial evidence was inconclusive and probably inadequate in legal terms, and subsequent discussion of *engine similarity* lacked precision and statistical rigour. Regan [27] is addressing this lacuna and

[21] The CCA now bans general use of mobile/(ear/head)phones and even hearing aids.

[22] Searches were instigated in the cases of Varshavsky and Ejsmont. Two players have been expelled from tournaments; Singh's colleague Umakanth Sharma was banned for 10 years.

Table 2 summarises the percentage of *Move Matches* (MM) with engines' preferences for many of these scenarios. It does not yet show 'mean error' [16] but does serve as an effective sighting 'scope' to target scenarios with 'SRFA'.

Table 2. Frequency of player-engine Move Matches[23]

Player	Date	Pos.	MM%	Player	Date	Pos.	MM%
Ejsmont	2007-07	104	77.6	Azmaiparashvili	1995	465	61.7
Fischer	1970+	718	67.4	Allwermann	1998-12	285	61.1
D.P.Singh	2006-04	686	64.7	SuperGMs	2005+	8447	57.5
Varshavsky/1	2006-06	170	64.2	Varshavsky/2	2006-06	44	38.3

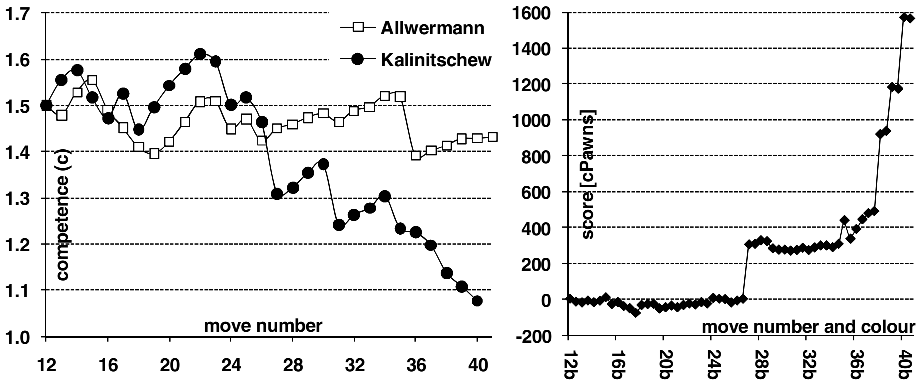


Fig. 3. Allwermann-Kalinitschew. Left: c- profile. Right: Game value in centipawns.

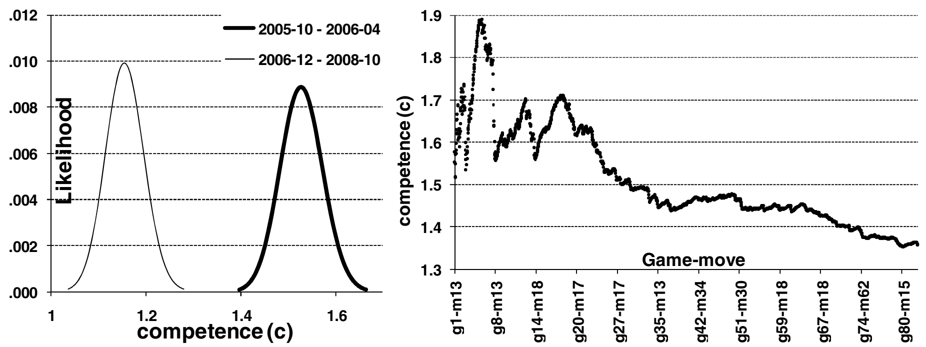


Fig. 4. D.P.Singh. Left: Probability Density Function of *apparent competence c* in two periods. Right: Evolution of *apparent competence c* based on game data available, 2005-10 to 2008-10.

[23] Varshavsky/1-2 reflects the play of this player before and after he delayed a search.

Fig. 3 addresses the performance of Allwermann and Kalinitschew in their game at the Böblingen tournament. On the left are the *c-loci* for both players, and on the right is the TOGA II v1.2.1 evaluation of the game in Pawns of advantage to White.

D.P.Singh's play came under suspicion in the second half of 2006. His *apparent competence c* profiles before and after this period are compared in Fig. 4 (left) with the evolution of his *c* alongside: the constituent games are positioned in a c_{DPS}-$c_{Opponent}$ space in Fig. 2. An application proposed here is a real-time *dashboard* (*c* plot and move series) to deter clandestine activity and to help focus the Tournament Director's forensic resources appropriately. A 'web community' implementation is feasible[24] and would also popularise chess by increasing spectator engagement and understanding.

5 The View Forward

This paper has defined and demonstrated a way of mapping decision-making behaviour into a benchmark space of agents, enabling skill to be measured in absolute terms, and future performance to be predicted.

The rating approach described here has obvious applications in identifying unexpected and possibly unwelcome behaviour. Business transactions are increasingly being carried out by/with electronic means and via the internet, facilitating the collection of evidence on the large scale necessary to reach accurate statistical conclusions. Betting markets are increasingly being monitored. The financial sector is likely to be subject to increased regulation after the collapse of trust in major institutions. The maintenance of national security increasingly seems to require the identification of patterns of electronic communication.[25]

The intention is that the Bayesian approach adopted here will be developed in several dimensions:

- Bayesian results -v- patterns of *nth-preference choice* [15],
- richer computational architecture: datastore, parallelisation, job-control,
- refined $C(v)$ & $L(w)$ functions giving better SRFP benchmark spaces,
- comparison of Bayesian results with 'average error' results [16], and
- application of the approach in one or more non-game domains.

We invite interested readers to join us in using this Bayesian approach to skill assessment, performance prediction, and behaviour positioning.

Acknowledgments. The web has enabled this work to benefit from the previous initiatives of many of its contributors. We particularly thank 'SHREDDER' Stefan Meyer-Kahlen and colleague Eiko Bleicher for their continual help. We thank all those who assisted in any way, including John Beasley, Ian Bland, Paul Janota, Kerry Lawless, Soren Riis, and Jeff Sonas.

[24] Trusted on-web engines send evaluations to an event server which highlights excellent play.

[25] See, e.g., the UK (RIPA) Regulation of Investigatory Powers Act (2000), the USA Patriot Act (2001) and European Community Directive 2006/24/EC on Data Retention.

References

1. Elo, A.: The Rating of Chessplayers, Past and Present. Arco (1978)
2. Haworth, G.M^{c}C.: Reference Fallible Endgame Play. ICGA J. 26(2), 81–91 (2003)
3. Haworth, G.M^{c}C.: Chess Endgame News. ICGA J. 28(4), 243 (2005)
4. Haworth, G.M^{c}C.: Gentlemen, Stop Your Engines! ICGA J. 30(3), 150–156 (2007)
5. Di Fatta, G., Haworth, G.M^{c}C., Regan, K.: Skill Rating by Bayesian Inference. In: Proc. IEEE (CIDM) Symposium on Computational Intelligence and Data Mining, pp. 89–94 (2009)
6. Nalimov, E.V., Haworth, G.M^{c}C., Heinz, E.A.: Space-Efficient Indexing of Endgame Databases for Chess. In: van den Herik, H.J., Monien, B. (eds.) Advances in Computer Games, IKAT, Maastricht, The Netherlands, vol. 9, pp. 93–113 (2001)
7. Jansen, P.J.: KQKR: Awareness of a Fallible Opponent. ICCA J. 15(3), 111–131 (1992)
8. Donkers, H.H.L.M.: Nosce Hostem: Searching with Opponent Models. Ph.D. dissertation, Univ. of Maastricht (2003)
9. 'Euclid':[26] Analysis of the Chess Ending King and Queen against King and Rook. In: Freeborough, E. (ed.), Kegan Paul, Trench. Trubner & Co. (1895)
10. Nunn, J.: Secrets of Pawnless Endings, 2nd revised edn., pp. 49–69 (2002)
11. Conway, E.J.: Browne's Triumph. Chess Voice 12(2), 10 (1979)
12. Larkins, J.: Queen vs Rook: A Point for Our Side. Chess Voice 12(2), 11 (1979)
13. Stenberg, W.: Beer in the Ear. Chess Voice 12(2), 8–10 (1979)
14. Kopec, D.: Man-Machine Chess: Past, Present and Future. In: Belzer, J., Kent, A., Holzman, A.G., Williams, J.G. (eds.) Encyclopedia of Computer Science and Technology, vol. 26, pp. 230–270, 241–243, `http://tinyurl.com/8faahk`
15. Regan, K.W.: Measuring Fidelity to a Computer Agent (2007), `http://www.cse.buffalo.edu/-~regan/chess/fidelity/`
16. Guid, M., Bratko, I.: Computer Analysis of World Chess Champions. ICGA J. 29(2), 65–73 (2006)
17. Regan, K.W.: (2009), `http://www.cse.buffalo.edu/~regan/chess/computer/compare/Tournaments/-Toga10Crafty12Results.txt`
18. Meyer-Kahlen, S.: The SHREDDER chess engine (2007), `http://www.shredderchess.com/`
19. Friedel, F.: Cheating in Chess. In: Advances in Computer Games, Institute for Knowledge and Agent Technology (IKAT), Maastricht, The Netherlands, vol. 9, pp. 327–346 (2001)
20. Huber, R., Meyer-Kahlen, S.: The UCI Protocol (2000), `http://www.shredderchess.com/down-load.html`
21. Guid, M., Pérez, A., Bratko, I.: How Trustworthy is CRAFTY's Analysis of Chess Champions? ICGA J. 31(3), 131–144 (2008)
22. Glickman, M.E.: Parameter estimation in large dynamic paired comparison experiments. J. Royal Stats. Soc., Series C (Applied Statistics) 48(3), 377–394 (1999)
23. Chessbase News: D.P.Singh: Supreme Talent or Flawed Genius? (2007-01-07)
24. Chessbase News: D.P.Singh has survived his 'Agni Pariksha' (2007-03-01)
25. Greengard, M.: Cheating Hearts Redux. The Daily Dirt Chess Blog (2006-07-07)
26. TWIC: Tadeusz Gniota Memorial Tournament Report. This Week in Chess (2007-07-20)
27. Regan, K.W.: Player-engine choice-matching data (2009), `http://www.cse.buffalo.edu/~regan/-chess/fidelity/FLM.html`
28. Berger, J.N.: Theorie und Praxis der Endspiele, p.175 (Revised 1922) (1890)

[26] First identified by Berger [28] as Alfred Crosskill (1829-1904) who also analysed KRBKR.

Plans, Patterns, and Move Categories Guiding a Highly Selective Search

Gerhard Trippen

The University of British Columbia
Gerhard.Trippen@sauder.ubc.ca

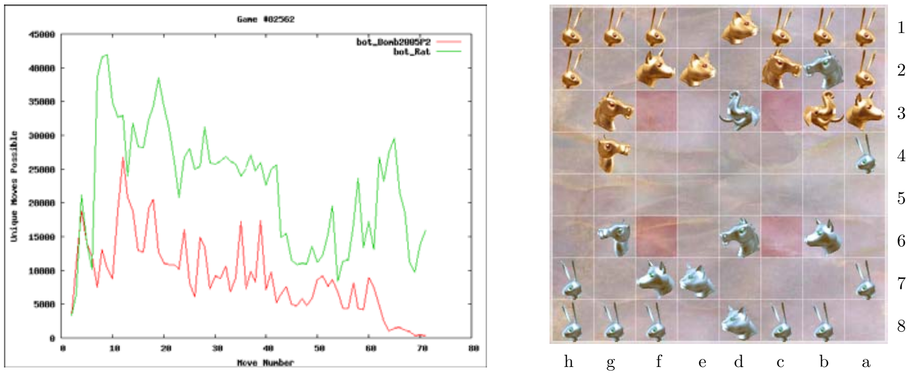
Abstract. In this paper we present our ideas for an Arimaa-playing program (also called a *bot*) that uses plans and pattern matching to guide a highly selective search. We restrict move generation to moves in certain move categories to reduce the number of moves considered by the bot significantly. Arimaa is a modern board game that can be played with a standard Chess set. However, the rules of the game are not at all like those of Chess. Furthermore, Arimaa was designed to be as simple and intuitive as possible for humans, yet challenging for computers. While all established Arimaa bots use alpha-beta search with a variety of pruning techniques and other heuristics ending in an extensive positional leaf node evaluation, our new bot, RAT, starts with a positional evaluation of the current position. Based on features found in the current position – supported by pattern matching using a *directed position graph* – our bot RAT decides which of a given set of plans to follow. The plan then dictates what types of moves can be chosen. This is another major difference from bots that generate "all" possible moves for a particular position. RAT is only allowed to generate moves that belong to certain categories. Leaf nodes are evaluated only by a straightforward material evaluation to help avoid moves that lose material. This highly selective search looks, on average, at only 5 moves out of 5,000 to over 40,000 possible moves in a middle game position.

1 Introduction

Arimaa is a modern board game designed to be difficult for computers [1]. It was invented by Omar Syed and Aamir Syed, and was motivated by the defeat of former world Chess champion Garry Kasparov by a Chess-playing computer developed by IBM called DEEP BLUE in 1997. Syed and Syed have offered a prize of USD 10,000 until 2020 to the first person, company or organization to develop a program that can defeat three selected human players in an official Arimaa match [2].

Arimaa is a two-player partisan board game, i.e., Gold can only move gold pieces, and Silver can only move silver pieces. All information about a position is known to both players at any time. There are no chance moves involved, such as moves based on randomization generated by dice. The game can be played with a standard Chess set. Each player has (in descending order of strength) an elephant, a camel, two horses, two dogs, two cats and eight rabbits. The letters representing those pieces in the game notation are e, m, h, d, c and r, for the silver pieces. For the gold pieces we use capital letters. The game is won by moving a rabbit to the *goal* rank, which means to bring it to the opponent's base row, but it can also be won by immobilizing all of the opponent's pieces, or by capturing all his rabbits. Below we briefly discuss four well-known problems in Arimaa.

The first problem that makes the game difficult for computers is that a player is allowed to use up to four steps in a single turn. A step consists of moving a piece to an adjacent square. It is also possible to push or pull an opponent's piece with a stronger piece. For example, in Fig. 1(b) the gold elephant could move from b3 to b4 pulling the silver horse from b2 to b3. Thus, a push or pull requires two steps. In a single turn players can move up to four different pieces, and this generates a huge number of possible moves. There are about 2,000 to 3,000 moves possible in the first turn depending on the way a player chooses to set up his pieces, and during the middle game the number of possible moves ranges from about 5,000 to over 40,000 (see also Fig. 1(a)) – compared to an average of about 30 for Chess.



(a) (b)

Fig. 1. (a) Graph of possible number of unique moves, generated with Brian Haskin's (aka Janzert) Game Grapher [3]. The peak moves possible for Silver was on turn 9 with 41939 moves possible. **(b)** Position on turn 9 for Silver in game #82562.

The 4-step moves make it quite difficult for computer programs to perform a deep search. Looking ahead only two moves for both players (= four plies) means to search to a depth of 16 steps. In the Arimaa Computer World Championship and in the Arimaa Challenge against humans, the time per move is restricted to two minutes. Simple alpha-beta bots are currently not able to reach this depth under the tournament settings nor in any reasonable amount of time. Even 12 steps might already be too deep. Only through extensive pruning, a variety of other heuristics, and quiescence search, are bots able to search deeper to find better moves.

To see whether bots perform better if they are given more time or are allowed to search deeper unrestricted by time, a P3 (= 3 plies = 12 steps) version of David Fotland's former Arimaa World Champion bot BOMB was available for a short while. However, the average move time in most of the games played was over 30 minutes, and at times the bot needed hours to find a move. Although it is basically impossible for a human to predict accurately the next 20 steps, it is still reasonable to look ahead three moves and obtain a good idea of the resulting position. Also the very simple bot ARIMAASCORE – which only needs about 1 second per move when playing at the P2 (= 2 plies) level – needs, on average, 5 minutes per move when playing at the P3 level.

The non-defined initial setup of the pieces is a second problem for bots. While CHESS programs can resort to a huge opening database, the initial setup of the 16 pieces is not fixed in Arimaa, which makes it very difficult to generate an opening database.

The third problem is formed by endgames. Endgames of the kind found in CHESS, in which only a few pieces remain, are also very rare in Arimaa. In many games a great number of pieces are still on the board when a rabbit reaches the goal.

The fourth problem area for computers are captures. Captures are performed not by moving onto an opponent's square like in Chess, or by jumping, as in CHECKERS. Instead, if a piece lands on a "trap" square (there are four of them: c3, f3, c6 and f6 (see Fig. 1(b)), and there are no pieces of the same color on any of the four squares adjacent to the trap, then the piece is captured and removed from the board. So, while in CHESS a piece might be captured in a single move by any stronger or weaker piece, in Arimaa often a weaker piece must be pushed and pulled towards home traps (c3 and f3 for Gold) by a stronger piece, where it can finally be captured. More details of such a plan will be given in Section 2.

Several research papers presenting bots have been published. David Fotland presented his World Champion Arimaa Program at the Computers and Games Conference in 2004 [4]. Haizhi Zhong and Christ-Jan Cox both wrote a Master's Thesis on Arimaa several years ago [5,6].

All the bots described above, and other well-established Arimaa bots, use an iterative deepening alpha-beta search with a variety of pruning techniques and other heuristics ending in an extensive positional leaf node evaluation. Quiescence search and other enhancements add to the strength of those programs.

Upper Confidence bounds applied to Trees (UCT) [7] and related research (e.g., [8,9]) have shown great success in the domain of the classical board game GO. Several members of the Arimaa community have discussed the usefulness of some of these approaches for Arimaa [10]. Jeff Bacher, the programmer of the current Computer World Champion program CLUELESS, is one of several people who also implemented or started to implement an UCT bot. Tomas Kozelek is also working on a UCT bot for his Master's Thesis [10]. Currently, it seems the traditional alpha-beta bots are still more successful. Pure random playouts seem not to be useful. In games where one player is missing an elephant (the strongest piece) and another player is missing a rabbit, the higher percentage is won by the party with the missing elephant. This contrasts with our intuition that the player with the elephant should win most of the games.

Our new bot Rat follows an approach different from UCT and from the traditional alpha-beta search. Rat follows a more human way of thinking by first analyzing the position, finding suitable plans, and then trying a certain highly selective number of moves. A tactical search, alpha-beta with some search extensions, ensures that the moves do not lead to too great a loss of material, by evaluating the leaf nodes with a straightforward evaluation function considering only the material of both players. A conceptually similar approach, the Technology Chess Program, was presented by James Gillogly nearly 40 years ago [11]. Jonathan Schaeffer presented a related approach, Planner, that determines a long-range strategy based on an assessment of the current position, and makes moves in the short-term that are consistent with a long-term objective [12].

However, the details of our bot Rat are different and the main concepts are explained in this paper, which is organized as follows. Section 2 introduces an often-used strategy (especially against Bomb and other bots) known as the *elephant-horse attack* (EH attack). Bomb and some other bots, or earlier versions of them, try to take the horse hostage close to their own trap. However, this often leads to decentralization of the elephant, and if the bot plays very passively afterwards, then kidnapping and capturing of several of the bot's pieces might be possible. We will call this *flash-kidnapping.* Section 3 shows how our bot "thinks", i.e., how Rat makes a positional evaluation of the current position and prioritizes plans which result in an ordered list of moves. We also describe move categories that drastically limit the number of possible moves for any given position. Section 4 explains the use of *directed position graphs* for pattern matching by giving an example graph for the EH attack. Section 5 gives a brief overview of the general performance of our bot. The paper concludes by ideas for future research in Section 6.

2 Elephant-Horse Attack, Horse Hostage, and Flash-Kidnapping

To motivate the idea of using plans, in this section we present a common strategy against bots before we discuss details of the implementation of our bot in the next section. One strategy for attacking an opponent's trap is to advance the elephant together with a horse on the same wing. We illustrate the strategy from the view point of Gold. To attack the c6 trap the gold elephant should be positioned on d6, keeping it close to the center and able to switch quickly to the other wing if necessary. The gold horse should occupy b6 so that the trap will be enclosed from both sides. Often Silver's elephant returns to c5 to protect the trap, and even more often the elephant might push Gold's horse to a6 or b7 to take it hostage. This is exactly the position our bot Rat is waiting for to follow the flash-kidnapping strategy (explained below). However, before starting this strategy it is important for Gold to secure its own traps. Gold's Southeast (f3) trap will be guarded by the camel on g3. In this way Silver's smaller pieces will not advance on the East wing. Also, the Silver's camel does not see a target on this side and rarely starts an attack there. Gold's Southwest (c3) trap will be protected by a horse and a dog. (See also Fig. 1(b) with switched colors.)

After creating this horse hostage situation RAT tries to follow a strategy that we call *flash-kidnapping*. The whole sequence takes a maximum of 40 steps, i.e., 10 plies, given the "cooperation" of the opponent. With the great branching factor of Arimaa – on average there are over 17,000 possible moves in any given position [2] – it is not expected that a bot will be able to detect such a sequence without having knowledge of this plan.

Figure 2 (a) shows the initial situation without the East wing. In the first move Gold pulls a Silver victim closer to her elephant (see Fig. 2 (b)). The intended capture of this victim lies still 6 plies ahead, so it is basically impossible for Silver to see. However, many bots try to avoid a situation where a weaker piece stands next to the opponent's elephant. If they do not, then in the next move Gold's elephant can flip the victim from d7 to d5 by first pulling it and then pushing it south (see Fig. 2 (c)). The advantage of this flip is that the elephant is back on d6 building a barrier against other Silver pieces that might want to help and free the victim, because now the intentions of Gold's elephant become quite clear. But the capture still lies four plies ahead, so it remains difficult for Silver to detect. Thus, the silver elephant might continue to hold the horse hostage.

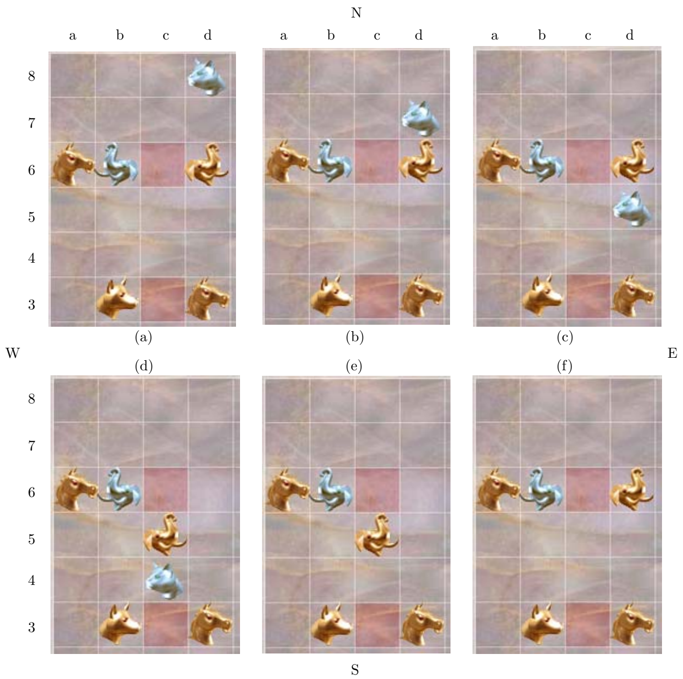


Fig. 2. Flash-kidnapping takes 10 plies (= 40 steps)

A double push brings the victim to c4 (see Fig. 2 (d)). Saving the victim with its elephant is often not desirable for Silver because it leaves the Northwest trap (c6) too weak. So, in the next move, Gold can capture the victim (see Fig. 2 (e)). And another move later, Gold's elephant can return to d6 to repeat the whole procedure (see Fig. 2 (f)).

3 Plans and Move Categories

Generally, alpha-beta bots generate a huge variety of possible moves in a given position and search recursively down to a certain depth at which the position will be evaluated. Our bot Rat imitates a more human approach. Before generating any moves, a positional evaluation of the current position is performed, which determines which plans Rat should follow next. Thus, rather than generating all possible moves in any given position, Rat uses a highly selective search. Before UCT became very popular in the recent years, many Go programs generated moves and evaluated them to find good candidate moves [13], because for Go, a brute-force search approach seemed unpromising. We follow a similar approach. In the following we will give a detailed explanation of Fig. 3.

3.1 Positional Evaluation and Plans

Algorithm Positional Evaluation and Algorithm Add Plan. The positional evaluation starts with the location of certain pieces. Trap control is particularly important for Arimaa. Based on the location of pieces and the number of pieces on the board Rat also tries to identify the game phase (opening, middle game, endgame). The priority of the plans is given through the structure of the evaluation function. This means there is a hard-coded order given through our implementation. Moves belonging to certain plans will be appended to a singly-linked list. However, while going through the function and collecting additional information, flags will be set that lower the priority of plans, and plans might be added later instead. While this is not done in our actual implementation, the list of plans could be implemented through a priority queue.

The first plan considered is a 4-step goal search, because if a player's rabbit reaches the goal he wins the game. However, this plan will not be added to the list if rabbits are not close enough to the opponent's home rank. Next, captures are considered, but if the opponent has the possibility of capturing a stronger piece than we can, defenses are tried first. Therefore, we first look at the opponent's possible captures to see which pieces or traps must be defended to avoid captures. Based on the outcome of both our possible captures and the opponent's, a decision is made which of the two plans (capture or defense) should be appended to the list first. After this, plans based on the outcome of the pattern matching (see Section 4) are added. Currently, only one strategy is considered, the flash-kidnapping strategy described in the previous section. Here a great number of additional strategies could be added by generating more directed position graphs that we use for pattern matching. In the order given below, the

```
Algorithm Alpha-Beta Search
    Input: plans tried
        and usual parameters like position, $\alpha, \beta, \ldots$
    Output: move and score
    If depth $\geq 2$ and quiet then
        score = simple eval; return score;
    list of plans Q = Positional Evaluation
    in the usual alpha-beta loop
        dequeue next plan from Q;
        Generate Move (plan) and make it;
        recursive call;
        compare score to bounds and update;

Algorithm Generate Move
    Input: a plan
    Output: a move
    generate all moves in a move category
        corresponding to the plan;
    sort the list of moves;
    return best move;

Algorithm Positional Evaluation
    Input: plans tried
        and parameters like position, depth, ...
    Output: ordered list of plans, Q
    Q += Add Plan(goal, ...)
    Q += Add Plan(capture, ...)
    If depth < 2 then
        Q += Add Plan(pattern matching, ...)
        Q += Add Plan(frames, ...)
        continuing with hostages, forks,
        camel hunt, retreats, trap defenses,
        goal preparations, flips, basic attacks,
        clearing of traps, elephant centralization

Algorithm Add Plan
    Input: a plan to consider and plans tried
        and position, its features, ...
    Output: a plan
    If all conditions for plan met
        then return plan; else return null;
```

Fig. 3. Main functionality of algorithms used to find a move

evaluation function adds the following plans to the priority list: frames, hostages, forks, hunt the camel, retreats, trap defenses (if not necessary earlier because of possible captures), goal preparations, flips, basic attacks, clearing of traps, and finally elephant centralization. This order is mostly fixed although some of the plans might not be added depending on the current position (see Algorithm Add Plan) or some flags might switch some of the priorities. Further refinement is certainly necessary here. However, this sequence of basic ideas will be used in particular if no other plan can be found through pattern matching. We believe that the pattern matching module will have the greatest impact on improving the playing strength of our bot. We have tested this only for the flash-kidnapping strategy so far.

Algorithm Alpha-Beta Search. RAT performs an alpha-beta search to a depth of 2 plies. In the first call the function will generate a sorted list of plans for the player starting with an empty list. Similarly to the Chess program PARADISE presented by David Wilkins [14], RAT considers which plans have been tried (and possible refuted) earlier in the search. This helps further reduce the branching factor in the tree search. However, the nature of Arimaa with four steps per move requires the player to be able to work on different plans within the same move. For example, the first two steps might be used to defend a home trap while the next two steps might be used to attack an opponent's trap. Therefore, all plans that have not yet been tried in a higher level of the tree are considered in a recursive call. When a recursive call of the alpha-beta search reaches the opponent, RAT will use the same positional evaluation function to generate the opponent's list of plans. Quiescence search will be performed in case of captures only. The alpha-beta search has a straightforward leaf evaluation function. It checks whether a rabbit reached a goal, and it calculates the value of the material on the board. Only if plan A leads to higher material gain (or lower material loss) than a plan that appears earlier in the ordered list, plan A is chosen over the other plan. The search ends after a certain time limit is reached.

To summarize the generation of plans, we can say that RAT is much more limited than, for example, PARADISE and Jacques Pitrat's CHESS COMBINATION PROGRAM [15] that both use production rules to produce plans. RAT knows only a certain set of plans embedded in the positional analysis. The order is mostly fixed although a number of flags and decisions based on features found in the position can lead to a certain reordering.

3.2 Move Categories

As described above, the plans are assigned different priorities and this yields an ordered list of plans, which itself translates into an ordered list of move categories for which actual moves must be generated.

RAT knows only a very limited number of move categories including goals, rabbit advancements, captures, retreats, kidnappings, attacks, and some trap protections corresponding to the plans mentioned above. Most of the trap protections and some of the trap attacks are implemented through patterns. This means RAT checks whether it can reach a certain local pattern from a given position. RAT knows a few hundred of these local patterns.

Algorithm Generate Move. When RAT generates moves of a certain category, all moves of this type are listed and immediately evaluated. Generally speaking, the overall resulting position is not evaluated, but rather the location of the moved pieces. For example, in the category rabbit advancement, the closer a move brings a rabbit to its goal the higher the score this move will receive. Often, trap defense should use stronger pieces to prevent the opponent from surrounding the trap by simply pushing weaker pieces aside, and furthermore, to avoid exposing weaker pieces to the opponent. For example, a cat or a dog on the side of a trap could easily become a victim of a horse that pulls it to the opponent's half of the board. Captures also include the value of the piece captured to determine the ranking. Only the top three moves in each category are considered strong enough, and will therefore be used as possible candidates in the alpha-beta search. This can also be restricted to the best move.

3.3 Two Other Details of the Implementation

One of the key facts making Arimaa difficult for computers is the use of 4-step moves. It was mentioned in Section 3 that it might be necessary or useful to split the four steps to follow two different plans. In particular, a fourth step that is not necessary to follow a certain plan involving an attack of an opponent's trap might be used to help protect a home trap. This fourth step may not be necessary, but is used as a fill-in step considering that the home trap might be under attack later in the game. Usually, there are several possibilities for such a step and this number increases considerably if there are two steps left for the trap protection plan. Including these fill-in steps in the regular alpha-beta search can make the tree much wider than necessary. If the trap is not under attack and in need of immediate defense we do not add the plan of trap protection into the alpha-beta search. Instead, we add a quick search after our main line has been determined and fill in some trap protecting moves afterwards.

A note about time management. The alpha-beta search is only used to confirm that certain moves that follow a particular plan do not lead to loss of material, or, e.g., that our bot Rat can indeed win material. The priority of the plans is solely determined by the positional evaluation. Therefore, if the bot has tried at least one plan already, i.e., has searched the whole subtree with the opponent's responses, and none of them could refute the plan, then Rat will end the search earlier if that helps accumulate reserve time.

4 Pattern Matching with Directed Position Graphs

Within the positional evaluation of the current position, Rat additionally uses pattern matching to determine the most promising move ordering. To store the positions, we use a data structure rather similar to a *directed acyclic word graph* (DAWG). A DAWG is a trie, i.e., a prefix tree, that eliminates prefix redundancy and also suffix redundancy. This data structure is space efficient and the lookup time is proportional to the length of the search string. We use a *directed position graph* (DPG), which is allowed to be cyclic. We further reduce the required space by eliminating infix redundancy.

When a human player studies an Arimaa position, the exact location of minor pieces such as dogs, cats, and rabbits is often not considered important if they are in a quadrant that is currently not under attack or involved in a plan. Those are the pieces that Mikhael Botvinnik might call Type III or Type IV pieces [16]. Therefore, when trying to match a position we focus on the most important features first. As an example of how to use DPGs for pattern matching we built a DPG for the horse hostage setup and the ensuing flash-kidnapping. It consists of approximately 250 nodes. A simplified version is shown in Fig. 4. The DPG will be searched using breadth first search (BFS).

First, when playing Gold, the location of Silver's elephant is matched. Based on this, we learn whether we actually need to mirror all the following locations, because although we always refer to the c6 trap, the hostage situation could also take place at the f6 trap. Next, we try to match the location of Gold's elephant. Then we check whether Gold's horse is held hostage by Silver's elephant. If Silver's elephant stands on b6 (eb6), then possible locations for Gold's horse are a6 or b7 (Ha6 or Hb7). Also, Gold's elephant could stand on d6 or c5 (Ed6 or Ec5) (see also Fig. 2), and we would still recognize this position as horse hostage. This means that in our DPG directed edges can lead from eb6 to Ed6 and Ec5, and then from both of those to both Ha6 and Hb7 (see Fig. 4). Thus, we have four different combinations that we would identify as horse hostage situation. This is greatly simplified compared to the actual DPG that Rat is using. Before starting the flash-kidnapping, the home traps should be sufficiently secured by

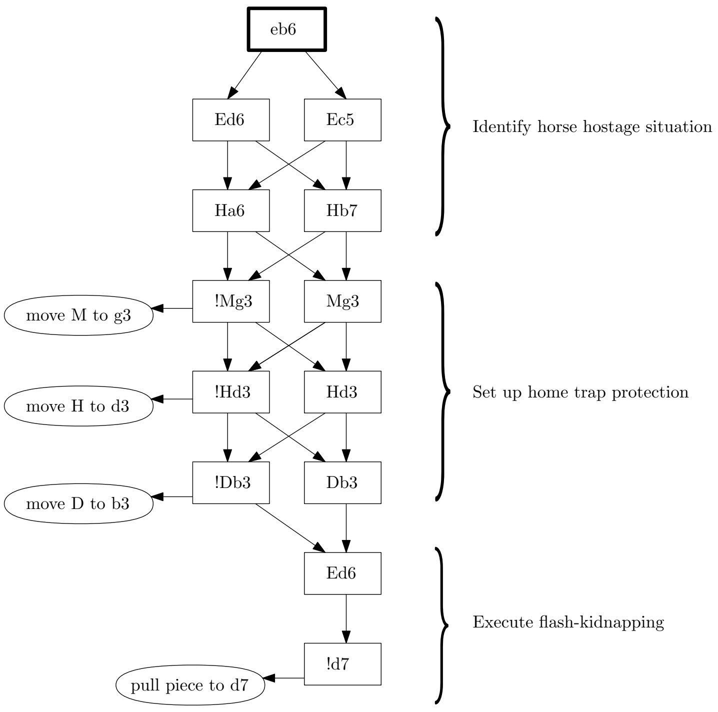


Fig. 4. Simplified DPG for flash-kidnapping strategy: When the BFS reaches a round node, this move will be added to the move list. Capital letters are used for Gold, small letters for Silver. Piece types are given using the capitalized bold letter in the words **E**lephant, ca**M**el, **H**orse, **D**og, **C**at, **R**abbit. A ! notation means no piece a of certain type if the type is specified, otherwise no piece at all.

camel, horse, and dog. So, from the nodes Ha6 and Hb7, edges are leading to a node Mg3. However, there are also edges leading to a node !Mg3, which indicates that Gold's camel is not on g3. A child of this node is then a leaf with a command to move the camel to g3. Both Mg3 and also !Mg3 lead to Hd3 and also !Hd3, while the latter has a child commanding a horse to d3. Because we search the DPG using BFS we can see that it is more important to bring the camel to g3 than to bring the horse to d3. We continue down with Db3 and !Db3. The positions of these three pieces form eight possible infixes. The logical sequence of first recognizing horse hostage, then setting up camel, horse, and dog, and finally executing the flash-kidnapping suggested that we should match the location of Gold's elephant once more to determine which move should be added to the list of candidates. This also greatly simplifies the task of editing the DPG.

As we use BFS to traverse the DPG, the DPG need not be acyclic. Indeed in some situations we might prefer the camel to stand on e3 instead of g3. So in our full DPG we also have nodes Me3 and !Me3. An edge is leading from !Mg3 to !Me3 and vice versa. The camel might not be able to reach g3 in one move (or remaining steps thereof) because it is too far West. But it may be able at least to reach e3, moving it closer to its intended destination.

5 Experimental Results

The Arimaa website [2] allows human players as well as bots to choose from a huge number of bots to play against. Most participating bots from former Computer World Championships are available with a wide variety of settings; restricted in search depth as P1 or P2, or playing with a certain time limit, for example, Blitz and Fast; or also with the original Computer Championship (CC) setting (unrestricted search depth, 2 minutes per move). Rat has beaten several of them. Most of the P1 bots and several of the P2 bots can be beaten regularly. Rat can also beat generally stronger bots, even if they play in the CC setting, if these bots take the horse hostage and Rat can use the pattern matching to follow the flash-kidnapping plan. Currently, no UCT bots are regularly available as they are still under development or have been considered too weak by their developers to participate in the Championship. Therefore, our results are restricted to the common alpha-beta bots. Overall, Rat is still a weak bot and cannot play well in many situations because the knowledge of how to play or what plan to follow in these positions is simply missing. We hope that by adding more DPGs, i.e., "teaching" more strategies, Rat or any other bot could be transformed into a stronger bot.

6 Future Work

Bot Rat is the first Arimaa bot that strictly uses move categories. A positional evaluation of the current position prioritizes plans and leads to the ordering of move categories and determines which of those can be generated. Rat performs a highly selective alpha-beta search with only a material evaluation at the leaf node level. Traditional Arimaa bots generate a huge number of possible moves – many of which a human would never consider at all – and then evaluate the leaves with a complex evaluation function. Due to the high branching factor of Arimaa, even with a great amount of pruning, only a depth of at most 20 steps in search extensions seems reachable with current computers. Many strategies require looking much further ahead. However, Arimaa is still a young game and it is not clear how deep those strategies could be. An interesting idea for future research might be to extract strategies from existing games automatically with the help of a computer program.

So far we have implemented only one single strategy to demonstrate our approach. Many more strategies could be added to create a stronger program. Knowledge of various strategies can be added using space-efficient directed position graphs. A plain text DPG can be edited easily without actually modifying the source code of the program. While we used a DPG to follow a certain strategy in the middle game, DPGs could also be used in the opening and in the endgame.

This exploration delved forty years into the history of chess programming in the hope that these approaches might lead to more successful Arimaa bots than the popular alpha-beta searchers. However, further examination is needed to determine whether this approach or other useful techniques, like UCT for example, could help go beyond the strength of these.

Acknowledgments. We would like to thank the anonymous referees for their careful review and incisive and encouraging comments. We greatly appreciate their constructive criticism that helped improve the paper. Thanks to the ACG conference participants for their additional questions and remarks.

References

1. Syed, O., Syed, A.: Arimaa - a new game designed to be difficult for computers. International Computer Games Association Journal 26, 138–139 (2003)
2. Syed, O.: Arimaa - the next challenge (2009), http://arimaa.com/arimaa/
3. Haskin, B.: Arimaa game graphs (2006), http://arimaa.janzert.com/gamegraphs
4. Fotland, D.: Building a world champion Arimaa program. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 175–186. Springer, Heidelberg (2006)
5. Zhong, H.: Building a strong Arimaa-playing program. Master's thesis, University of Alberta, Dept. of Computing Science (September 2005)
6. Cox, C.J.: Analysis and implementation of the game Arimaa. Master's thesis, Universiteit Masstricht, Institute for Knowledge and Agent Technology (March 2006)
7. Kocsis, L., Szepesvari, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
8. Coulom, R.: Computing Elo ratings of move patterns in the game of Go. ICGA Journal 30(4), 198–208 (2007)
9. Chaslot, G., Winands, M., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Progressive strategies for monte-carlo tree search. In: Wang, P. (ed.) Proceedings of the 10th Joint Conference on Information Sciences, Salt Lake City, USA, pp. 655–661 (2007)
10. Syed, O.: Arimaa forum (2009), http://arimaa.com/arimaa/forum
11. Gillogly, J.J.: The technology Chess program. Artificial Intelligence 3, 145–163 (1972)
12. Schaeffer, J.: Long-range planning in computer Chess. In: ACM 1983: Proceedings of the 1983 annual conference on Computers: Extending the human resource, pp. 170–179. ACM, New York (1983)
13. Müller, M.: Computer Go. Artificial Intelligence 134(1-2), 145–179 (2002)
14. Wilkins, D.: Using patterns and plans in Chess. Artificial Intelligence 14, 165–203 (1980)
15. Pitrat, J.: A Chess combination program which uses plans. Artificial Intelligence 8, 275–321 (1977)
16. Botvinnik, M.M., Brown, A.: Computers, Chess and Long-Range Planning. Springer-Verlag New York, Inc., Secaucus (1970)

6-Man Chess and Zugzwangs

Eiko Bleicher and Guy Haworth

School of Systems Engineering, University of Reading, UK
`bleicher@k4it.de`,
`guy.haworth@bnc.oxon.org`

Abstract. With 6-man Chess essentially solved, the available 6-man Endgame Tables (EGTs) have been scanned for zugzwang positions where, unusually, having the move is a disadvantage. Review statistics together with some highlights and positions are provided here: the complete information is available on the ICGA website. An outcome of the review is the observation that the definition of *zugzwang* should be revisited, if only because the presence of *en passant capture* moves gives rise to three new, asymmetric types of zugzwang.

1 Introduction

Six-man Chess is essentially solved and the Nalimov [1] Endgame Tables (EGTs) to DTM, Depth to Mate, have been widely promulgated for some time [2][1]. The corpus of perfect information is a challenge to datamine for both helpful guidelines and for the pathological positions – the deep, exceptional, bizarre, and amusing positions.

A *zugzwang* position is defined here as one where the side to move would prefer that it were the other side's turn to move. They are remarkable in themselves and the 'zug', much sought after by composers, is a running theme in the Study community [3-9]. The zug is also the counterexample to the assumption of the Null Move Heuristic that having the move is an advantage. When 5-man chess was solved, the two sets of respectively DTM and DTC[2] [10] EGTs were searched [11] for zugs[3]. A later review of Thompson's 6-man pawnless DTC EGTs also included a zug search [12].

The zugzwang search reported here is almost entirely of the highly available Nalimov DTM EGTs [2,13]. The search was carried out via the web using Bourzutschky's GTBGEN and the first author's EGTs and JAVA code. Unpublished FEG DTM EGTs [14] for 5-1(p) chess and DTC EGTs for 6-man chess [15-16] exist and Bourzutschky has supplied [16] a zug review of 5-1(p) chess from the latter.

Section 2 considers *zugzwang* definitions and identifies three new types of zug: section 3 covers logistics. Section 4 is a summary of the main findings. Tables 1-5 list various illustrative statistics and positions; the full details are available via the ICGA website [17] which will host the evolving story of the zugzwang.

[1] *Essentially* because positions with castling rights are not yet included in EGTs.
[2] DTC ≡ Depth to Conversion, i.e., to force-change and/or Mate.
[3] Incidentally providing a partial cross-check of agreement between the two sets of EGTs.

2 Definitions of *The Zugzwang*

Zugzwang is defined [18] as *pressure to take action* and a *zugzwang position* is defined to be one where this pressure is unwelcome – where the first player would rather 'pass the position across'.[4] However, in [19] a *zugzwang position* is defined as 'a position in which whoever has the move would obtain a worse result than if it were the opponent's turn to play'. Note that this now involves the 2nd player's perspective and focuses only on the outcome without considering its achievement or likelihood. Other authorities refer to zugzwangs as *reciprocal* or *mutual zugzwangs*. The words *whoever, reciprocal,* and *mutual* suggest a symmetry, perhaps assuming incorrectly that the 2nd player can always pass back the 1st position to the 1st player.

Consider the *en passant zone EPZ* of Chess, i.e., those positions where there is an en passant capture option. Let $p1 \in$ EPZ: what now are positions $p2$ and $p3$? The proposal here[5] is to clarify this situation by formalising the notions of *passing over* or *losing the move* as one of *playing a null move* or *nulling*. Now the rules of chess define $p2$ and $p3$: the e.p.-capture option in $p1$ disappears if not played immediately, $p2$ does not feature any e.p.-capture option as the 1st player has not moved a Pawn two squares, and similarly, $p3$ is $p1$ without e.p.-capture option. Positions in EPZ are 0.62% of those in their endgame and zugs in EPZ comprise 0.22% of the total.

Let the *Level A* zugzwang, our focus here, be described in these terms:

a) positions are valued from 1st player's perspective: loss (0), draw (1), win (2),
b) if there is force-symmetry and/or no *e.p.*, 1st player is assumed to be White
   Black plays first in 11 Table 1 positions (Zs 04-5, 07-9, 12, 22-4, 26-27),
c) 1st player in position $p1$ (value $v1$) nulls to $p2$ (value $v2$) iff $v2 > v1$,[6]
d) 2nd player may *or may not*, cf. Table 1's Z08-9, null to $p3$, value $v3 \le v2$,
e) if $p3 \equiv p1$, $v3 \equiv v1$. If $p3 \neq p1$, $p3$ is 'stalemate' or $v3 \le v1$.

Fig. 1 is formatted so that the 1st player nulls 'to the right' and the 2nd player nulls to the left to increase value to themselves. Clearly, in addition to the three familiar zug types A1-3,[7] we now have, exclusively in the EPZ, just three more types A4-6:[8]

1) type A1 ≡ '121' ≡ 'draw-win-draw', q.v. Z01 in Table 1
2) type A2 ≡ '010' ≡ 'loss-draw-loss', q.v. Z02: A1-3 are *no net gain* for player 1
3) type A3 ≡ '020' ≡ 'loss-win-loss', the *full-point* zug, q.v. Z03
4) type A4 ≡ '120' ≡ 'draw-win-loss', a *net loss* for player 1, q.v. Z05-6
5) type A5 ≡ '021' ≡ 'loss-win-draw', a *net gain* for player 1, q.v. Z07
6) type A6 ≡ '01(1)' ≡ 'loss-draw(-draw)', a *net gain* for player 1[9], q.v. Z08-9

Clearly, type A4-6 zugs are asymmetric. Considered only in terms of the first two positions, A4 becomes A1, A5 becomes A3 and A6 becomes A2.

---

[4] Not strictly possible, as in 'passing the position across', the side to move changes.
[5] Our project log notes that Bourzutschky proposed the *Null Move* concept on 2005-05-31.
[6] *Value* is calculated as normally, assuming that the option of a null move is not available.
[7] Types are distinguished by the sequence v_1-v_2-v_3 rather than just by the sequence v_1-v_2.
[8] Unless position $p3$ is *stalemate*, its value $v3 \le v1$ as the 1st player has one less move in $p3$.
[9] The 2nd player may even prefer to play on rather than stalemate their opponent.

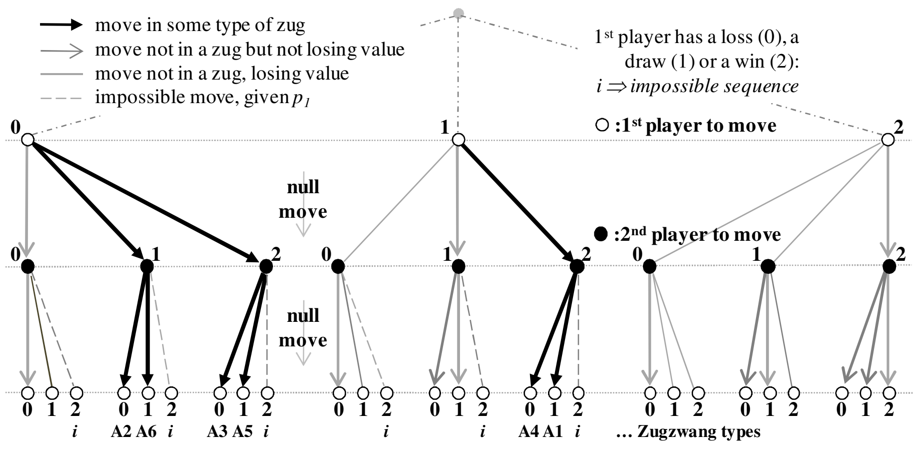


Fig. 1. The six types of Level A zugzwang: the familiar types A1-3 and the 'EPZ types' A4-6

Table 1. Some examples of zugzwangs of types A1-6[10,11]

			Zug		value and DT*x*				
id	**Endgame**	**Position**	**Type**		**p1**	**p2**	**p3**	***x***	**Flag**
Z01	KPK	1k6/1P6/2K5/8/8/8/8/8 w	A1	121	=	+2	=	Z	
Z02	KBKP	8/8/8/8/8/1pK5/kB6 w	A2	010	-1	==	-1	Z	m
Z03	KPKP	8/1pK5/kP6/8/8/8/8/8 w	A3	020	-1	+1	-1	Z	m
Z04	KPPKP	8/8/8/3k4/2pP4/2K5/1P6/8 b - d3	A2	010	-25	=	-15	M	
Z05	KBPKPP	8/8/8/1p6/1Pp5/8/4K3/2kB4 b - b3	A4	120	=	+21	-20	M	u
Z06	KPPKPP	8/1p6/1k6/pP6/K7/P7/8/8 w - a6	A4	120	=	+21	-30	M	m
Z07	KP(5)KP(4)	8/8/8/2p5/1pP1p3/kP2P3/Pp1P4/1K6 b - c3	A5	021	-0	+1	==	Z	
Z08	KRPKPP	8/8/8/8/pP6/p7/k1K5/1R6 b - b3	A6	011	-0	=	==	Z	u
Z09	KPPPKP	8/8/8/8/1pP5/kP6/P7/K7 b - c3	A6	011	-0	==	==	Z	u
Z10	KRNKNN	8/8/8/8/2n5/1n6/R1N5/3K1k2 w	A2	010	-1	=	-1	M	
Z11	KRBNKNN	8/8/8/8/2n5/1n6/R1N5/1B1K1k2 w	A3	020	-1	+38(+)	-1	M	
Z12	KBPKNP	2n5/8/8/2B5/1Pp5/k1K5/8/8 b - b3	A2	010	-18	=	-18	M	i
Z13	KQBKNP	8/1K6/8/2k5/Q1Bn4/8/2p5/8 w	A1	121	=	+21	=	M	U
Z14	KQNKBN	8/8/8/8/8/nkb5/1N6/Q1K5 w	A1	121	=	+35	=	M	U
Z15	KQNKNN	8/8/8/n1Q5/2n5/8/2kN4/K7 w	A1	121	=	+35	=	M	U
Z16	KQRKBB	8/8/6b1/2R5/3K4/4Q3/5b2/5k2 w	A1	121	=	+26	=	M	U
Z17	KQRKRB	2r5/R1Q5/b7/8/8/2K5/8/1k6 w	A1	121	=	+17	=	M	U
Z18	KQBPKB	1k6/1P6/K7/1Q6/B1b5/8/8/8 w	A1	121	=	+9	=	M	U
Z19	KQBPKP	8/8/8/8/1p6/1PQ5/kBK5 w	A1	121	=	+11	=	M	U
Z20	KRRRKQ	8/8/8/8/4q1k1/2R5/1R1K3R w	A1	121	=	+22	=	M	U
Z21	KBPKNP	3n4/8/8/5pP1/8/8/8/1kBK4 w - f6	A1	121	=	32	=	M	
Z22	KBPKNP	8/8/3B2n1/K7/1pP5/k7/8/8 b - c3	A2	010	-32	=	-32	M	
Z23	KNPKPP	8/8/8/p7/Pp6/3N4/3K4/k7 b - a3	A2	010	-25	=	-25	M	
Z24	KRPKPP	8/1K6/1p6/2k5/1pP5/8/8/2R5 b - c3	A2	010	-21	=	-21	M	
Z25	KBPPKP	1BK5/8/k7/Pp6/8/P7/8/8 w - b6	A1	121	=	+17	=	M	
Z26	KNPPKP	8/8/K7/PNk5/Pp6/8/8/8 b - a3	A2	010	-23	=	-23	M	
Z27	KPPPKP	8/8/1k6/1P6/KPp5/8/P7/8 b - b3	A2	010	-17	=	-17	M	

[10] Values from 1st player's perspective: + win, = draw, == stalemate, - loss.

[11] Flags: *m* maxDT*x*, *i* inaccessible, *s* symmetric, *u* unique of type, U unique in endgame.

3 Enumeration: Endgames and Zugzwang Occurrences

These notes explain the lexical ordering of men and of endgames, and the principles used for counting the occurrences of zugzwangs.

The men are listed in the *strength order* K-Q-R-B-N-P. White has at least as many men as Black. In *m-m* endgames, White's *lead men* are at least as strong as Black's.

No attempt is made to eliminate unreachable positions in EGT statistics: this is usual as there is no general algorithm. With this limitation, the count is of FEN-distinct and functionally unique zugzwangs. Thus, no zugzwang can be physically transformed into any other. The following subtleties should be noted.

a) For force-symmetric zugs $z \notin$ EPZ, type A1 and A2 zugs are equivalent:
 the count of A2 zugs is shown in brackets.
b) Given force symmetry, A3 zugs usually appears in two physical forms:
 - the two physical versions were identified[12] and counted as one here.
c) When both Kings are on a1-h8 or a8-h1 in pawnless endgames:
 Nalimov has both physical versions of the position if there are two;
 the two physical versions were identified and counted as one here.
d) When e.p.-capture has been enabled but is actually illegal, q.v. Z12:
 the position is counted here as different from that without the *e.p.*,
 1st player would have to realize that the e.p.-option is illusory,
 FIDE's recently reworded Article 9.2 now seeks to ignore the *e.p.* [20-22].
e) An example of an unreachable zug:
 position Z12 also implies the prior 1. b2-b4, impossible on two grounds:
  before 1. b2-b4, the side not to move, Black, is in (double) check,
  check from a Pawn on its home square is itself impossible,
  therefore, the position prior to Z12 is also unreachable.

We note one small, historical and now resolved hiatus with respect to these results. The identification of zugs of types A4-6 was a serendipitous accident[13] and was initially regarded as a bug by GTBGEN'S author Marc Bourzutschky.

When the 'last 16' 3-3p endgames to KPPKPP were published, both by MB converting his FEG EGTs to Nalimov's format[14] [23] and by Nalimov returning a disc to Hernandez [24], MB discovered that although he had anticipated a 2-byte format for the KQPK(B/R)P EGTs in GTBGEN, Nalimov had in fact discovered that the 1-byte format would suffice [25]. MB realigned with Nalimov and removed the ability to detect type A4-6 zugs before a GTBGEN was provided that could address these two EGTs. MB now advises [16] that no type A4-6 KQPKBP or KQPKRP zug exists.

Results for Level A zugs have been published as a set of files [17]. Table 2 provides summary statistics for the six blocks of 6-man zugs, 3-3(p), 4-2(p) and 5-1(p). There are 293 5-1p zugs, 261 created by a touring Knight being on the wrong foot.

Table 3 details the occurrences of e.p. zugs including the new types of zug. Table 4 lists in lexical order all endgames with A3 zugs, together with an example of their

[12] The identification of symmetries and equivalences was done by the first author's code.
[13] The serendipitous accident has its place of honour in the history of discovery.
[14] After reverse-engineering the unpublished format of the FEG EGT data format.

deepest such zug. Table 5 is a miscellany of exemplar positions: an A2 e.p. zug with p3's depth less than p1's (P03), an A3 e.p. zug (N01), A2 and A3 zugs dezugged by the addition of an e.p. opportunity (N02-3) or castling option (P01), zugs unaffected (P02) or created (N04-8) by giving castling rights to the 1st or 2nd player, 5-1p zugs (P06-13), 7-man zugs (B01-6) and *Zugzwang Studies* (S01-12) featuring zugs. Elkies [26] composed N01-10 and Bourzutschky [15-16] found B01-6.

Table 2. Level A zugs: summary statistics for each 6-man endgame group[15,16]

	Item \ Group	**3-3**	**4-2**	**5-1**	**3-3p**	**4-2p**	**5-1p**	**Total**
01	**Endgames**	55	80	35	65	95	35	365
02	***No zugs***	12	50	35	12	29	28	166
03	No A1 zugs	12	50	35	12	30	28	167
04	No A2 zugs	31 (+ 9)	71	—	22 (+ 3)	51	—	175 (+ 12)
05	No A3 zugs	55	80	—	50	67	—	252
06	***A unique zug***	4	1	0	1	2	0	8
07	One A1 zug	4	1	0	1	4	0	10
08	One A2 zug	1	2	—	1	6	—	10
09	One A3 zug	0	0	—	6	3	—	9
10	**A1-A6 zugs**	27,597	20,017	0	380,363	478,682	293	906,952
11	A1 zugs	27,470	8,434	0	361,725	373,479	293	771,401
12	A2 zugs	127	11,583	—	15,543	105,069	—	132,322
13	A3 zugs	0	0	—	2,700	133	—	2,833
14	A4 zugs	—	—	—	394	0	—	394
15	A5 zugs	—	—	—	0	0	—	0
16	A6 zugs	—	—	—	1	1	—	2

Table 3. Statistics for the fourteen endgames with *e.p.-zugs*

	e.p.-zugs		**A1**		**A2**		**A3**		**A4**		**A5**		**A6**	
Endgame	**wtm**	**btm**	**wtm**	**btm**	**wtm**	**btm**	**wtm**	**btm**	**wtm**	**btm**	**wtm**	**btm**	**wtm**	**btm**
KPPKP	35	20	35	0	0	20	0	0	0	0	0	0	0	0
KBPKBP	10	——	10	——	0	——	0	——	0	——	0	——	0	——
KBPKNP	130	5	130	2	0	3	0	0	0	0	0	0	0	0
KBPKPP	18	18	18	4	0	13	0	0	0	1	0	0	0	0
KNPKNP	156	——	156	——	0	——	0	——	0	——	0	——	0	——
KNPKPP	250	19	250	17	0	2	0	0	0	0	0	0	0	0
KPPKPP	1,301	——	869	——	39	——	0	——	393	——	0	——	0	——
KQPKQP	75	——	72	——	3	——	0	——	0	——	0	——	0	——
KRPKBP	20	7	20	0	0	7	0	0	0	0	0	0	0	0
KRPKNP	27	14	27	0	0	14	0	0	0	0	0	0	0	0
KRPKPP	0	2	0	0	0	1	0	0	0	0	0	0	0	1
KBPPKP	1	0	1	0	0	0	0	0	0	0	0	0	0	0
KNPPKP	0	1	0	0	0	1	0	0	0	0	0	0	0	0
KPPPKP	8	2	8	0	0	1	0	0	0	0	0	0	0	1
Totals	**2,031**	**88**	**1,596**	**23**	**42**	**62**	0	0	**393**	**1**	0	0	0	**2**

[15] (+*n*) indicates that there are *n* endgames whose A2 zugs merely mirror their A1 zugs.

[16] The eight zugzwangs which are unique across their endgames are Z12-Z19 in Table 1.

Table 4. The 43 endgames with *full-point* type A3 zugs: maxDTM examples[17,18,19]

			val / DTM			Total	
id	**Endgame**	**Position**	**p1**	**p2**	**p3**	**Depth**	**Flag**
F01	KBPKNP	k7/Bp4n1/1K1P4/8/8/8/8/8 w	-25	+16	-25	41	
F02	KBPKPP	8/8/8/3K1k1p/3P2p1/6B1/8 w	-57	+28	-57	85	
F03	KNPKNP	8/8/8/8/K1k5/P1p5/n1N5 w	-13	+24	-13	37	u
F04	KNPKPP	8/8/8/4kp1p/3N4/2KP4/8 w	-71	+26	-71	97	
F05	KPPKPP	8/8/8/p4p2/k1P5/2K1P3/8 w	-17	+103	-17	120	c.f. '18'
F06	KQNKQP	8/8/3N4/8/k2p4/3q4/KQ6 w	-14	+6	-14	20	u
F07	KQNKRP	QN6/Kpk5/1r6/8/8/8/8/8 w	-8	+32	-8	40	u
F08	KQPKQP	8/8/8/1Pq5/8/1K1Q4/5p2/2k5 w	-24	+95	-24	119	
F09	KQPKRB	2K1k3/2P5/8/8/8/8/1r2b3/4Q3 w	-20	+15	-20	35	u
F10	KQPKRP	8/8/8/1Q6/8/1pP5/2k2r2/K7 w	-12	+42	-12	54	
F11	KRNKNP	8/8/8/8/p7/N2k4/RK1n4 w	-1	+28	-1	29	u
F12	KRNKPP	8/8/8/2N5/8/7p/k5pR/2K5 w	-33	+17	-33	50	
F13	KRPKBP	8/8/8/8/2p5/2Pk4/1KR1b3 w	-15	+30	-15	45	
F14	KRPKNP	8/8/8/3n4/k7/p1P5/K1R5 w	-1	+33	-1	34	u
F15	KRPKPP	8/8/6R1/k1P5/2K5/7p/6p1/8 w	-71	+9	-71	80	
F16	KBBPKQ	8/8/8/8/1K6/BBP5/8/qk6 w	-102	+13	-102	115	
F17	KBNPKB	K7/P1k5/8/8/8/8/6N1/5B1b w	-2	+14	-2	16	
F18	KBNPKN	BK1n4/NP1k4/8/8/8/8/8/8 w	-1	+32	-1	33	
F19	KBPPKB	K7/P1k5/8/8/8/8/6P1/5B1b w	-2	+14	-2	16	
F20	KBPPKP	8/8/8/8/3k4/1K1p4/1P3P2/B7 w	-18	+16	-18	34	
F21	KBPPKQ	3K1kq1/8/4PB2/3P4/8/8/8/8 w	-34	+40	-34	74	c.f. '19'
F22	KBPPKR	8/8/8/8/2k5/P1P5/rBK5 w	-18	+22	-18	40	
F23	KNNPKN	K1k5/P2N4/4N3/3n4/8/8/8/8 w	-1	+19	-1	20	
F24	KNPPKN	K7/P1kN4/8/3P4/n7/8/8/8 w	-2	+20	-2	22	
F25	KNPPKP	8/8/8/K7/P1k5/1p6/3P4/4N3 w	-20	+23	-20	43	
F26	KNPPKQ	1K1k2q1/8/2P5/3N4/8/2P5/8/8 w	-20	+45	-20	65	
F27	KNPPKR	N1k5/2P5/rPK5/8/8/8/8/8 w	-22	+12	-22	34	
F28	KPPPKN	n7/P1k5/K7/PP6/8/8/8/8 w	-2	+13	-2	15	
F29	KPPPKP	8/8/8/5k2/3K1p2/3P3P/3P4/8 w	-20	+19	-20	39	
F30	KPPPKQ	k7/q1PK4/P7/8/8/2P5/8/8 w	-15	+19	-15	34	
F31	KPPPKR	1K6/1P1k4/1r6/1P6/2P5/8/8/8 w	-21	+36	-21	57	
F32	KQNPKN	QN6/KP6/8/nk6/8/8/8/8 w	-1	+8	-1	9	
F33	KQNPKQ	8/8/8/5N2/1q6/8/Q2P4/K1k5 w	-3	+35	-3	38	
F34	KQPPKQ	8/8/8/1q6/5P2/P7/Q7/K1k5 w	-4	+33	-4	37	
F35	KRBPKB	K7/P1k5/8/8/8/8/6R1/5B1b w	-2	+14	-2	16	
F36	KRBPKN	RK6/B3n3/1Pk5/8/8/8/8/8 w	-2	+14	-2	16	u
F37	KRBPKP	8/8/8/8/8/1k6/pP6/BRK5 w	-11	+21	-11	32	
F38	KRBPKQ	1qk5/8/RBP5/8/8/8/8/1K6 w	-41	+13	-41	54	u
F39	KRNPKN	8/8/8/8/n7/P7/K1k5/RN6 w	-1	+24	-1	25	
F40	KRNPKQ	1K3N2/3R1P2/1kq5/8/8/8/8/8 w	-11	+16	-11	27	
F41	KRPPKN	K7/P1k5/R1P5/8/2n5/8/8/8 w	-1	+20	-1	21	
F42	KRPPKQ	3R4/q7/k1P5/P7/K7/8/8/8 w	-36	+12	-36	48	u
F43	KRPPKR	8/8/8/8/rPK5/1RP5/2k5 w	-26	+29	-26	55	

[17] The depth of a type A3 or A5 zug is defined as the sum of the depths of *p1* and *p2*.
[18] KPPKPP zug F05 is the deepest A3 zug with *dtm* = 120.
[19] KBPPKQ zug F21 has the maximal DTM-depth 'shallower side' loss (here) with *dtm* = 34.

Table 5. More didactic positions including 5-1p zugs and e.p. and/or castling effects[20]

				DTx				
id	**Endgame**	**Position**	**Type**	**p1**	**p2**	**p3**	***x***	**Flag**
P01	KQNKRR	1KQNk2r/7r/8/8/8/8/8/8 w - -	A1	=	+54	=	Z	c
P02	KRRKRB	r3kb2/1RK4R/8/8/8/8/8/8 w - -	A1	=	+7	=	Z	c
P03	KPPKPP	8/6p1/4k1P1/4Pp2/3K4/8/8/8 w - f6	A2	-24	=	-19	M	e
P04	KNNKP	7k/8/5NK1/7p/8/8/N7/8 b - -	B2	-0	1	-0	Z	B2
P05	KPPKPP	8/8/p2k4/6p1/3K2P1/P7/8/8 b - -	C	=	=	=	Z	C
P06	KBPPPK	5kBK/5P1P/7P/8/8/8/8/8 w - -	A1	=	+2	=	Z	5
P07	KBPPPK	7K/5kBP/5P1P/8/8/8/8/8 w - -	A1	=	+2	=	Z	5
P08	KBPPPK	8/B1k5/K7/P7/P7/P7/8/8 w - -	A1	=	+3	=	Z	5
P09	KBPPPK	1k6/8/KP6/BP6/1P6/8/8/8 w - -	A1	=	+1	=	Z	5
P10	KNPPPK	7K/5k1P/4N2P/7P/8/8/8/8 w - -	A1	=	+2	=	Z	5
P11	KPPPPK	5k2/5P2/4K3/7P/7P/7P/8/8 w - -	A1	=	+2	=	Z	5
P12	KQNPPK	4k1KQ/5NPP/8/8/8/8/8/8 w - -	A1	=	+2	=	Z	5
P13	KRPPPK	1k6/1P6/K7/RP6/P7/8/8/8 w - -	A1	=	+2	=	Z	5
N01	KPPPKPPP	8/1p6/8/pP4pK/5kP1/P7/8/8 w - a6	A3	-1	+1	-1	Z	e
N02	KPPKP	3K4/8/3k4/8/3Pp3/4P3/8/8 b - -	A2	-4	=	-4	Z	e
N03	KPPKPP	8/8/8/5pK1/4kPp1/8/7P/8 b - -	A3	-1	+1	-1	Z	e
N04	KQP(6)KRRBP(3)	Q1K1k2r/PPP1p2p/b1r1P2P/2p5/2P5/8/8/8 w k -	A2	-1	=	-1	Z	c
N05	KQP(6)KRRBP(5)	Q1K1k2r/PPP1p2p/bprpP2P/2p5/2P5/8/8/8 w k -	A3	-1	+2	-1	Z	c
N06	KQP(8)KRRBP(7)	Q1K1k2r/PPP1p2p/bprpP1pP/2p5/2P2pP1/8/5P2/8 b k g3	A3	-2	+2	-2	Z	c e
N07	KRBNP(3)KRBP	r3k1KR/3p2PB/3P2N1/3P3b/8/8/8/8 w q -	A3	-1	?	-1	Z	c
N08	KRBNP(4)KRBP(3)	r3k1KR/3p2PB/2pP2N1/7b/1pP5/8/1P6/8 b q c3	A3	-?	+1	-?	Z	c e
N09	KPPKP	8/8/8/3k2P1/4pKP1/8/8/8 w - -	B1	+84	+25	+84	M	B1
N10	KPPKP	8/8/3k4/1K1p4/1P6/1P6/8/8 b - -	C	=	=	=	Z	C
B01	KNNNKNN	7k/8/4N3/4NN2/n2K4/8/8/3n4 w - -	A1	=	+17	=	Z	s
B02	KRBBKQB	8/8/8/2b2q2/B7/1R3B2/2k1K3 w - -	A3	-96	+2	-96	Z	u
B03	KRBBKQN	8/5B1q/6R1/3n4/8/8/2KB4/k7 w - -	A3	-6	+2	-6	Z	
B04	KRRRKRR	8/8/8/8/3Rr3/kr6/2KRR3 w - -	A1	=	+2	=	Z	U
B05	KBBBBKQ	6B1/1B4qB/5k2/8/3K4/8/6B1/8 w - -	A2	-35	=	-35	Z	s
B06	KBBNNKQ	8/8/8/4q3/2k4N/5B2/N1K2B2 w - -	A3	-7	+41	-7	Z	u
S01	KRKN	8/8/2k1K3/8/3R4/4n3/8 w - -	[9] #2, ar-Razi (~850)					
S05	KQKRP	1rk5/8/8/3Q4/8/1p6/1K6/8 w - -	[9] #6					
S03	KRKBN	k1K5/2n5/8/8/b7/1R6/8/8 w - -	[4] #457, [9] #5, Nunn					
S05	KRKBN	k3b3/n1K5/R7/8/8/8/8/8 w - -	[9] #5a					
S06	KNNKP	8/8/1p6/1K6/2N5/3N4/8/k7 w - -	[9] #9					
S07	KNPKP	8/8/8/6Pk/4K3/4N2p/8/8 w - -	[9] #7					
S08	KNPKN	8/8/8/5KPk/8/8/8/5N1n b - -	[9] #7a: A2 zug					
S09	KNPKP	8/8/8/4k3/7p/P3K1N1/8 w - -	[9] #8					
S10	KNPKN	8/8/8/4k3/8/P3K3/5N1n w - -	[9] #8a					
S11	KNPKPP	8/8/8/5p2/4k1p1/4N1P1/5K2 w - -	[39] #5.1, Mandler					
S12	KPPKPP	8/5pk1/8/2p1PK2/2P5/8/8/8 w - -	[9] #4					

Positions S01-S12 are from studies where White wins in an essentially unique way. They all feature a level A zug in both a *try* and the mainline solution and are mainly taken from Beasley [9]. The appendix can accommodate only a few of the solutions so there are plenty of exercises here for the reader.

[20] *c* ≡ zug-significant castling rights given to 1st or 2nd player, B/C ≡ Level B/C zug, *e* ≡ e.p. significant, and *5* ≡ 5-1p zug.

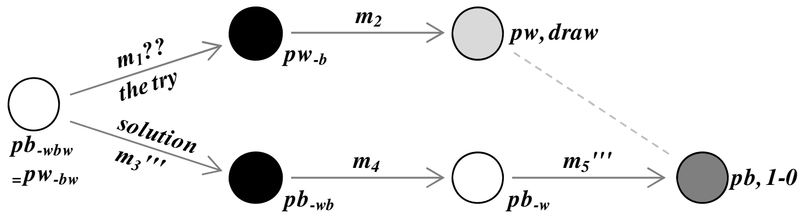


Fig. 2. The *Zugzwang Study* scenario

4 Commentary: Statistics, Gems, and Studies

Our EGT search has identified a corpus of over 900,000 zugs which may be reviewed in statistical terms, datamined for gems of various sorts, and put in the context of the Chess Study, the chess-engine and the game itself.

In addition to the Level A zug defined above, two further levels of zug are notable, q.v. Table 5. At *Level B*, a B1 (B2) zug is merely inconvenient, requiring the winner (loser) to make more (less) moves in some metric before some defined goal. Positions P04 (B2), N09 (B1) and S01-S12 feature examples. Note that the number of *moves to goal* may not be affected in all metrics.[21] At *Level C*, zugs do not impact value or depth in any metric but the side to move would rather play a null move: Regan [27] identified P05 and Elkies [26] identified N10, both draws in which a null move eases the task of the defender. The likelihood of a win in a theoretically drawn position can be modelled using the concept of a Reference Fallible Player [28-32].

4.1 The Statistics

Pawnless zugs are just 5.26% of the total; pawnful zugs account for the vast majority. There are more pawnful than pawnless positions but the presence of at least one pawn increases the density of zugs by a factor of four. Zugs are also more frequent when Knights, parity-bound and unable to *lose the move*[22], are present.

Type A3 zugs are clearly rarer than A1-2 zugs and have been the focus to date: it had been established that no pawnless A3 zug exists with 5 men or less [11].

The presence of an *en passant* feature in a zug has not attracted attention so far, perhaps for three reasons. Type A1-3 zugs in EPZ remain zugs of the same type if the en passant opportunity is removed. Secondly, e.p. constrains two Pawns to 14 of the 2,256 positions available and, at 0.22% of the total as in Table 3, e.p. zugs are few indeed. Finally, the very few type A4-6 zugs had not been discovered.

The next challenges are to trawl for zugs in small EGTs for positions with castling rights, and in large 7-man EGTs. Under the DTZ_{50} metric which recognizes the 50-move Rule, some zugs, q.v. Table 4, lose their status or change type.

[21] KNNKP position P04: *dtz* = *dtc* = 4m if a null move is available; *dtm* = 4m regardless.

[22] Alternating as they do between white and black squares.

4.2 Datamining for Gems

The deepest zugs come closest to not being zugs at all: the shallowest zugs feature the greatest change of advantage achievable by the null move. Relative depths may vary with the depth-metric chosen.

Rare gems have intrinsic value; zugs may be absolutely or type-unique within their endgame and/or have some rare feature. There are only 51 type A2 e.p. zugs in which position *p3* is shallower than position *p1*: 20 in KPPKP and 31 in KPPKPP.

The absence of a pawnless 5-man A3 zug naturally led to a search for one in 6-man chess. In response to the so-called *Pawnless Trébuchet Challenge* [33], Elkies had conjectured that position Z10, identified three years earlier [8], might just be one such: he did not claim [8] that it was as incorrectly announced by Roycroft [34]. The search turned to 6-man endgames [12], [33], [35] but the authors confirmed earlier in the work reported here that there were none [29]. Evidence of Elkies' remarkable prescience [36] is that Z10 perhaps comes closest of all 6-man pawnless positions to being an A3 zug. This is the only position found in which the 2nd player has to avoid the loss by first playing four unique draw-saving moves.[23]

With KRBNKNN Z11 derived from KRNKNN Z10 [8], the *A3 challenge* became one of reducing the number of Knights in such a zug. The zugs B06 [29], [37], B03 [16] and B02 [38] feature two, one and no Knights respectively though B02 leaves a residual challenge by requiring *obtrusive*, i.e., obviously-promoted, force.

Surprisingly, there are no A3 e.p. zugs in 6-man chess but Elkies recalled an 8-man example (N01) derived from an actual game [39]. Other than the 393 A4 KPPKPP zugs, there are just three A4-6 zugs: one A4 (Z05), no A5 zugs[24] and two A6 zugs (Z08-9). The A4 zug is unique in that the value of position *p3* is worse than the value of position *p1*, but the 1st player is a net winner in A5-6 zugs: the 2nd player's perspective is irrelevant in A6 zugs. Elkies has provided the first known A5 zug (Z07) and examples N02-3 of an added e.p. opportunity dezugging a zug.

Castling rights have not been included in EGTs. However P01 and P02 are the first known zugs where added castling rights dezug or not. Elkies [26] provided exemplar zugs (N04-8) where the provision of 0-0(-0) castling rights to the 1st or 2nd player creates a zug: some also feature a significant e.p. opportunity.

4.3 Zugzwangs and Studies

In the Chess Study, White is by convention challenged to draw or win. The appearance of a zugzwang position in a study is notable in itself and, if it is Black to move, suggests that White is just one ply from missing its objective. Mandler's study S11 [40] requires White to revisit a previous physical position 11 plies later but with Black to move: that position is therefore a Level B zug.

A *Zugzwang Study* is defined to be one in which the zugzwang not only appears in the main line of a study in White's favour but also appears as the refutation of a plausible sideline *try* [9]. Fig.2 illustrates the requirements for such a study: a position *p* must appear in its wtm form *pw* in the try and in its btm form *pb* in the main line; White's moves should be essentially unique and Black should play its 'best defence'.

[23] 1. … Nc5′ 2. Nd4 Ne3+′ 3. Kd2 Kf2′ 4. Ne6 {other moves pressure more} Nxe6′.

[24] And MB [16] reports that KPPPKPP* (assuming only P=Q allowed) has no A5 zugs.

Beasley gives some remarkable examples of the genre and his article on the theme, from which most of the study positions S01-S12 are taken, is recommended. The zugzwang study demonstrates that the aesthetic contribution [41] of a zugzwang position to a study must be judged in the context of that study and not in isolation.

5 Summary

The authors have searched the available Nalimov DTM EGTs for 6-man chess to identify all the Level A zugzwangs. Somewhat accidentally, we have discovered three new types of zugzwang to make six types in all: there are no other types.

Work will turn to zugs in the more recently arrived 6-man 'DTC' results [16] which will be compared with those of Nalimov[25] and Thompson [12], to Level B and C zugs, and to an examination of the occurrence of zugs in studies [42-43].

Complementing this review, the full results, including statistics, highlights and lists of all the zugs with their DTM depths, are published on the ICGA website [17]. The zugs may be studied using EGT query services on the web [13], [44] and we look forward to them being mined for gems by the Chess Studies community and others.

Acknowledgments. Our thanks first to Eugene Nalimov for his major contribution of 3- to 6-man DTM EGTs, to Andrew Kadatch for the software used to compress them, and to Marc Bourzutschky, Nelson Hernandez and Kyrill Kryukov for seeding their promulgation. Secondly, our thanks to Marc for his GTBGEN generalization of Nalimov's TBGEN used to identify the zugs, for his 5-1p and 7-man zugs and for his continued interest. Thanks go to Noam Elkies for a remarkable string of 'first exemplar' zugs involving e.p., castling or 5-1p chess. Finally, thanks to Ken Regan and Noam for the first known Level C zugs, and to John Beasley, Harold van der Heijden, Philip Henderson, Soren Riis and John Tamplin for stimulating dialogue.

References

1. Nalimov, E., Haworth, G.M^{c}C., Heinz, E.A.: Space-efficient Indexing of Endgame Tables for Chess. ICGA J. 23(3), 148–162 (2000)
2. Kryukov, K.: EGTs Online (2006), `http://kirill-kryukov.com/chess/tablebases-online/`
3. Roycroft, A.J.: Test Tube Chess: A Comprehensive Introduction to the Chess Endgame Study. Faber and Faber Ltd. (1972)
4. Nunn, J.: Secrets of Pawnless Endings, 2nd edn. B.T. Batsford, London (2004)
5. Nunn, J.: Secrets of Rook Endings, 2nd edn. B.T. Batsford, London. Gambit (1999)
6. Nunn, J.: Secrets of Minor-Piece Endings. B.T. Batsford, London (1995)
7. Beasley, J., Whitworth, T.: Endgame Magic. B.T. Batsford, London (1996)
8. Elkies, N.D.: No. 10965: mutual full-point zugzwang? EG 8-128, 320 (1998)
9. Beasley, J.: Creating reciprocal zugzwang studies. EBUR 12(2), 8–12 (2000)
10. Wirth, C., Nievergelt, J.: Exhaustive and Heuristic Retrograde Analysis of the KPPKP Endgame. ICCA J. 22(2), 67–80 (1999)

[25] First indications [31] are that the 'DTC' and 'DTM' statistics are in full agreement.

11. Haworth, G.M^{c}C., Karrer, P., Tamplin, J.A., Wirth, C.: 3-5-Man Chess: Maximals and Mzugs. ICGA J. 24(4), 225–230 (2001)
12. Tamplin, J., Haworth, G.M^{c}C.: Ken Thompson's 6-man Tables. ICGA J. 24(2), 83–85 (2001)
13. Bleicher, E.: Endgame Service based on Nalimov's EGTs (2009), `http://www.k4it.de/index.php-?topic=egtb&lang=en`
14. Tay, A.: A Guide to Endgame Tablebases, `http://www.horizonchess.com/FAQ/Winboard/-egtb.html` (2009)
15. Konoval, Y., Bourzutschky, M.S.: Private Communications (2007-8)
16. Bourzutschky, M.S.: Private Communications (2009)
17. ICGA: The ICGA website. Menu: Game-specific information – Western Chess - Endgames (2009), `http://www.icga.org`
18. Thyen, O., Clark, M., Scholze-Stubenrecht, W., Sykes, J.B.: The Oxford-Duden German Dictionary. OUP (2005)
19. Hooper, D., Whyld, K.: The Oxford Companion to Chess. OUP (1992)
20. Gijssen, G.: An Arbiter's Notebook: Monroi and Other Matters. Chesscafe.com (2006), `http://-www.chesscafe.com/text/geurt105.pdf`
21. Gijssen, G.: An Arbiter's Notebook: Interpreting the Rules. Chesscafe.com (2007), `http://www.-chesscafe.com/text/geurt110.pdf`
22. Gijssen, G.: An Arbiter's Notebook: Interpreting the Rules. Chesscafe.com (2009), `http://www.-chesscafe.com/text/geurt129.pdf`
23. Bourzutschky, M.S.: The 16 "missing" Nalimov files, `http://preview.tinyurl.com/aqevp2` (2006-07-18)
24. Hernandez, N.: 'Missing 16 received!'. Private Communication (2006-08-08)
25. Bourzutschky, M.S.: Tablebase comparison, `http://www.tinyurl.com/d3wny4` (2006-08- 10)
26. Elkies, N.D.: Private Communications (2009)
27. Regan, K.W.: Private Communications (April 2009)
28. Haworth, G.M^{c}C.: Reference Fallible Endgame Play. ICGA J. 26(2), 81–91 (2003)
29. Haworth, G.M^{c}C.: Chess Endgame News. ICGA J. 28(4), 243 (2005)
30. Haworth, G.M^{c}C.: Gentlemen, Stop Your Engines! ICGA J. 30(3), 150–156 (2007)
31. Di Fatta, G., Haworth, G.M^{c}C., Regan, K.W.: Skill Rating by Bayesian Inference. In: Proc. IEEE Conf. on Computational Intelligence and Data Mining, Nashville, USA, pp. 89–94 (2009)
32. Haworth, G.M^{c}C., Di Fatta, G., Regan, K.W.: Performance and Prediction: Bayesian Modelling of Fallible Choice in Chess. In: Proc. Advances in Computer Games. LNCS, vol. 12. Springer, Heidelberg (2009)
33. Roycroft, A.J.: Announcement of the Pawnless Trébuchet task. EG 7-116, 633 (1995)
34. Roycroft, A.J.: Report 1 on the Pawnless Trebuchet task. EG 7-117, 645 (1995)
35. Costeff, G.: EG 1-152 online (2009), `http://www.gadycosteff.com/eg/eg.html`
36. Beasley, J.: Gems Discovered by Computer. BESN Special Number 46, 6–7 (2005)
37. Bourzutschky, M.S., Konoval, Y.: 7-man Endgame Databases. EG 11(162), 493–510 (2006)
38. Haworth, G.M^{c}C.: Chess Endgame News. ICGA J. 29(2), 79 (2006)
39. Elkies, N.D.: On Numbers and Endgames: Combinatorial Game Theory in Chess Endgames. In: Nowakowski, R.J. (ed.) Games of No Chance. MSRI 29 (1996)
40. Beasley, J.: Depth and Beauty: The chess endgame studies of Artur Mandler (2003)

41. Iqbal, A.: A Discrete Computational Aesthetics Model for a Zero-Sum Perfect Information Game. University of Malaya, Kuala Lumpur, Malaysia (2008), http://metalab.uniten.edu.my/~azlan/Misc/phd_thesis_azlan_final.pdfPh.D.thesis
42. Costeff, G., Stiller, L.B.: Chess Query Language CQL (2003), http://rbnn.com/cql/
43. Van der Heijden, H.: Endgame Study Database III, 67,691 Studies (2005)
44. Tamplin, J.A.: Multimetric endgame service (2009), http://chess.jaet.org/endings/

Appendix: Some Zugzwang Lines

The 7-man lines are from Bourzutschky [16,37]. All moves are at least optimal moves given the move-selecting strategy nominated, and beyond that, the key is:

″″ ≡ only value-saving move (independent of move-choosing strategy),
″ ≡ the only optimal move, given the strategy nominated, e.g., SC$^-$M$^-$, and
° ≡ only move.

KPPKP Z04: positions *p1* and *p3* have different depths to mate.
p1, btm: {*dtm* = 25} SM$^-$/SM$^+$ 1. … cxd3″ 2. Kxd3″″ Kc5″ 3. Kc3″″ Kb5″ 4. Kb3″″ Ka5 5. Kc4″ Kb6″ 6. Kb4″″ Kc6 7. Ka5″ Kb7″ 8. Kb5″ Ka7″ 9. Kc6″ Ka8″ 10. b4 Kb8″ 11. Kb6″ Ka8″ 12. b5 Kb8° 13. Ka6″ Kc7″ 14. b6 Kc8″ 15. Ka7″″ 1-0
p2, wtm: 1. K~ Kxd4 = or 1. b~ cxb3″″ =
p3, btm: {*dtm* = 15} SM$^-$/SM$^+$ 1. … K(c6/d6/e6) 2. Kxc4″ {*dtc* = 7} 1-0

KBPKPP Z05: an A4, draw-win-loss, zug:
p1, btm: 1. … cxb3″″ 2. Bxb3″″ =
p2, wtm: SM$^+$Z$^+$/SM$^-$Z$^-$ 1. Ke1″ c3″ 2. Bb3 c2″ 3. K~ K(b1/d1/d2) 4. Bxc2″ Kxc2″″ 5. Ke1 K(b3/c3) 6. Kd1 Kxb4″″ 0-1
p3, btm: SM$^-$Z$^-$/SM$^+$Z$^+$ 1. … Kb1″ 2. Ke3″ Kc1″ 3. Be2 Kc2″ 4. Kd4″″ c3″ 5. Bd3+″ Kd2″ 6. Be4 c2″ 7. Bxc2″″ Kxc2″ 8. Kc5″″ Kb3 9. Kxb5″″ Ka3 10. Kc5 Kb3 11. b5″ 1-0

KRBNKNN Z11: an A3 zug adaption of Z10 which ‘just failed’ to be A3.
p1 ≡ *p3*, wtm: 1. N~ N(x)d3#″″; 1. R~ Nxb2#″″ 0-1
p2, btm: {“Black cannot maintain the bind” [8]} 1. … Nc5 2. Nd4 Ne3+ 3. Kc1 Kg1 4. Bf5 Ng2 5. Nf3+ Kf1′ 6. Bh3 Nd3+′ 7. Kb1 Nb2 8.Kxb2 {*dtc* = 1m, *dtm* = 8m}

KNNNKNN B01: the bK is boxed in but White must avoid a KNNK endgame.
p2, btm: {*dtz* = 17} SZ$^-$/SZ$^+$ 1... Ndc3 2. Nd7″″ Kh7 3. Nf4″″ Kg8 4. Nd6″″ Kg7 5. Ke5″″ Nd1 6. Kf5 Kh6 7. Kf6 Ne3 8. Nf7+″″ Kh7° 9. Ng5+″″ Kg8 10. Ne4″″ Kh7 11. Ne5 Nb6 12. Kf7″″ Nbc4 13. Nf6+″″ Kh6 14. Nf3″″ Nd6+ 15. Kf8 Nef5 16. Ng8+ Kh7° 17. Ng5+ Kh8° 18. Ng6# {10 of White’s 17 moves were unique winning moves} 1-0

KRBBKQN B03: an A3 zug with only one Knight.
p1, wtm: SZ$^+$/SZ$^-$ 1. Be8 Ka2 2. Bf7 Ka3 3. Bc1+ Kb4 4. Bd2+ Kc4 5. Be8 6.Ne7 0-1

{1. Bxd5 Qxg6+″″ {KBBKQ, *dtz* = 62) “is however probably *best defence*” [16]}.
p2, btm: SM^-Z^-/SM^+Z^+ 1. ... Qh3 2. Ra6+″ Qa3° 3. Rxa3#″ 1-0

KRKN S01: 1.Re3"" Ng1' 2.Kf5' (2.Kf4?? Kd4 pw =) Kd4' 3.Kf4"" pb 1-0

KRKBN S03: 1.Rb6"" Nb5' 2.Ra6+"" Na7+° 3.Kc7"" Be8' 4.Ra3"' (4.Ra2?? Ba4"" pw =) Ba4' 5.Ra2' pb 1-0

KPPKPP S12: 1.Kf4"" (1.Ke4?? Kg6" pw) Kg6' 2.Ke4"" pb Kg5' 3.e6"" fxe6' 4.Ke5"" Kg4 5.Kxe6"" 1-0

Solving Kriegspiel Endings with Brute Force: The Case of KR vs. K

Paolo Ciancarini and Gian Piero Favini

Dipartimento di Scienze dell'Informazione, University of Bologna, Italy
{ciancarini,favini}@cs.unibo.it

Abstract. Retrograde analysis is a tool for reconstructing a game tree starting from its leaves; with these techniques one can solve specific subsets of a complex game, achieving optimal play in these situations, for example a chess endgame. Position values can then be stored in "tablebases" for instant access, as is the norm in professional chess programs. While this technique is supposed to be only used in games of perfect information, this paper shows that retrograde analysis can be applied to certain Kriegspiel (invisible chess) endgames, such as King and Rook versus King. Using brute force and a suitable data representation, one can achieve perfect play, with perfection meaning fastest checkmate in the worst case and without making any assumptions on the opponent.

1 Introduction

In a zero-sum game of perfect information, Zermelo's theorem [1] ensures that there is a perfect strategy allowing either player to obtain a guaranteed minimum reward. In many games, discovering the perfect strategy seems to be synonymous with exploring a major portion of the game tree, which is unfeasible under current and foreseeable computer technology. Nowadays, it is possible to explore significant subsets of the game tree in such a way that, if a particular position is encountered during gameplay, its value has already been computed and the best strategy is immediately available. Most serious programs for playing chess include a so-called "endgame tablebase". Unlike opening books, the same tablebase can freely be used by any number of programs even under tournament conditions, on the basis that it contains no creative work but simply knowledge that rests on large amounts of processor time.

Currently, tablebases exist for all six-piece chess endings, with seven-piece positions in the process of being computed for the next few years. In many cases, the perfection of tablebase-powered play is unapproachable by even the strongest evaluation function, or indeed the strongest human player. Positions that most experts would have considered draws turn out to be mates in 300 or 500 moves [2], and seemingly hopeless games can be drawn by repetition. Tablebases are usually obtained through retrograde analysis. Analysis starts from final nodes, the leaves in the game tree corresponding to checkmates and stalemates, and then moves backwards in time to find out how those positions were obtained, until all positions of the desired type have been explored. The concept has been widely studied since the 1970s, so there is a large bibliography devoted to chess tablebases and their creation. We cite, for example, Bellman's seminal paper [3] and Stiller's systematic work on a series of chess endgames in [2].

The aim of this paper is to show that the same concept can be usefully applied to a game of imperfect information, as well, though it is only limited to finding situations where a player can force victory with probability 1. We use Kriegspiel (invisible chess) as an example. The game is identical to chess, except players can only see their own pieces and need to rely on messages from a referee to figure out where the opponent is. We give an algorithm for solving Kriegspiel endings that have so far only been approached with approximated or heuristic methods, and use it to build a Kriegspiel tablebase for the King and Rook versus King (KRK) ending.

The paper is structured as follows. In section 2, we describe Kriegspiel and summarize previous research in the field. Section 3 contains the actual algorithm, as well as considerations on its correctness, complexity, and optimizations. Section 4 is about the actual run of the algorithm and construction of the tablebase. Finally, conclusions and future perspectives are given in Section 5.

2 Kriegspiel

Kriegspiel is a chess variant invented at the end of the 19th century to make chess more like the 'war game' used by the Prussian army to train its officers. It is played on three chessboards in different rooms, one for either player and one for the referee. From the referee's point of view, a game of Kriegspiel is a game of chess. The players, however, can only see their own pieces and communicate their move attempts to the referee, so that there is no direct communication between them. If a move is illegal, the referee will ask the player to choose a different one. If it is legal, the referee will instead inform both players as to the consequences of that move, if any. Kriegspiel is not a standardized game, as there are several known sub-variants to the game; they differ in how much information the referee shares with the player with respect to pawn moves and captured pieces. Since our main concern in this paper is with a pawn-less endgame and a single capturable piece, the choice of ruleset is irrelevant. It suffices to remember that the referee will inform the players whenever a check or capture happens, when a move is illegal, and when the game ends.

The nature of Kriegspiel, being so similar to chess in some ways and yet completely different in others, caught the attention of some well-known computer scientists and was known and played at the RAND institute. Several endings have been studied, though so far always with the aid of heuristics or *ad hoc* considerations. For example, [4] deals with KPK using a set of directives and distinguishes between algorithmically won endings, which can always be won, and statistically won ones, wherein victory is only achieved with probability $1 - \epsilon$, with ϵ small (arbitrarily small in the absence of the 50 move rule). It is shown that certain instances of the KPK ending are of the former type, and some are of the latter. Ferguson studied two less common endings, KBNK in [5]

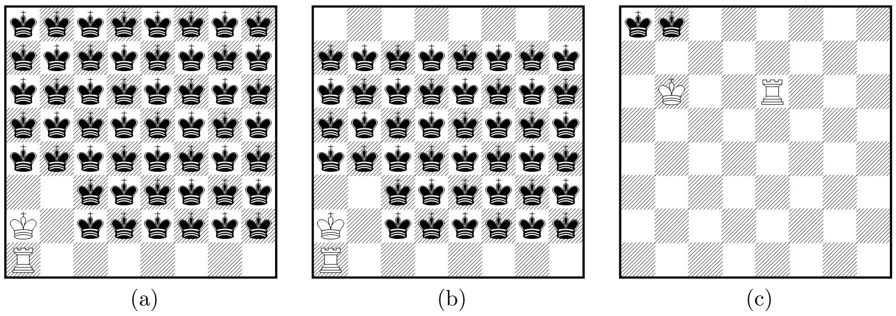


Fig. 1. (a) Highest uncertainty in KRK; (b) useless information: after two plies this board will be identical to the outcome of the same plies on (a); (c) mate in one

and KBBK in [6]. These can be won algorithmically, provided White can set up his[1] pieces in particular patterns and the black King is confined to certain areas of the board.

KRK is the most widely studied ending, probably because it is so simple in chess but not so simple in Kriegspiel, even though it is always won if White can secure his rook. Magari [7] gave an algorithm for solving KRK starting from a special position, and was the first to think of the black King as a ‘quantic wave’ whose actual location is not determined until the white pieces have moved. His definition would anticipate Sakuta’s research in [8], which dealt with invisible Shogi and introduced the idea of metapositions. Boyce [9] had previously given his own algorithm for solving KRK, expressed as a series of directives in the natural language and without any formalization. While Boyce’s conditions are more general than Magari’s, the problem of reaching the starting position remains. Neither algorithm is shown to be optimal for White. Finally, Bolognesi and Ciancarini used metapositions, *ad hoc* evaluation functions, and minimax-like tree search in [10] to solve KRK in the general case, showing it to perform better than Boyce’s directives. However, success with this method cannot be guaranteed without trying out every single case. As a side note, Shapley and Matros improvised a solution to KRK in Kriegspiel on an infinite board while attending the ninth Game Theory convention in 1998.

This topic is a subset of the more general problem of checkmating the opponent in Kriegspiel, either forcing the mate or maximizing one’s chance of doing so. Aside from Sakuta’s aforementioned work on Shogi, this direction was explored by Russell and Wolfe in [11], who focused on efficient handling of large belief states in Kriegspiel. Research by Amir, Nance and Vogel is marginally related in that they attempt to reconstruct the state of the board in [12]. The even more general problem of computer Kriegspiel is beyond the scope of this paper; we refer to such papers as [13] and [14] for two completely different views on the subject.

[1] For brevity, we use ’he’ and ’his’ whenever ’he or she’ and ’his or her’ are meant.

3 A Retrograde Analysis Algorithm

Let us formalize the problem as follows. $\mathbf{S}$ is the set of all possible game states (in this case, the set of legal chess positions limited to the KRK ending), and $i \in \mathbf{I} \subseteq P(\mathbf{S})$ is, at any given time, the information set for the player with the Rook, who will be assumed to be White from now on, and contains all possible game states at this point in time. Elements of $\mathbf{I}$ will often be called *metapositions* in this context and are easily represented by placing a black King on every square where the King might be; we will follow this convention throughout the paper. Players may choose moves $m \in \mathbf{M}$, the set of *pseudolegal* moves for the current information set. Pseudolegal means that the move is legal according to the rules of chess in at least one state of the current information set i, though it may turn out to be illegal when tried. More specifically, we can define a referee function $r : (\mathbf{I} \times \mathbf{M} \times \mathbf{S}) \rightarrow (\mathbf{I} \times \mathbf{R} \times \mathbf{S})$, where $\mathbf{R}$ is a set of referee messages, consisting of {`victory, draw, silent, check, illegal`}. A game state appears both as an input and an output, representing the unknown real state, which only the referee can access. The referee changes our knowledge of the board and also returns a message in response to a move. The message set would have to be expanded to handle additional endings, but it is ample enough to treat KRK, as only one check type is compatible with any move. Function r obviously depends on the black King's strategy.

Since the black King is the only unknown piece, it can readily be seen that any $i \in \mathbf{I}$ contains at most 52 elements; an example of this is given in Fig. 1, (a). While 52 states might not seem like a large amount, in order to provide an optimal solution to this endgame we need to be able to handle every possible instance of $\mathbf{I}$ optimally. Counting the subsets of the information set in Fig. 1 alone, we obtain $2^{52} - 1$ possibilities. Even using mirroring along the two axes and a diagonal, there are over 400 ways to place the white King and Rook, each allowing on average 40 to 50 positions for the black King, and as such 2^{40} to 2^{50} instances of $\mathbf{I}$. An approximate calculation leads to about 10^{16} or 10^{17} possible information sets, making the KRK ending in Kriegspiel not that far behind the whole game of checkers in sheer size. Bolognesi and Ciancarini [10] mention, among other things, these figures to motivate their decision to use a heuristic, approximated approach to the problem. Their problem does not fully overlap with ours, as they try to provide strategies for playing even when victory is not guaranteed; for example, in KRK, when the Rook does not start near the King and cannot immediately approach it. However, our algorithm is only concerned with algorithmically won positions, although it can be extended with heuristics to treat such a case.

When considering the size of the problem, it is clear that many of these Kriegspiel information sets are really redundant as they contain no information that can be exploited by the player in order to speed up the road to victory, at least in the worst case. Figure 1, (a)-(b) shows an example wherein removing states from the information set does not make any difference. If this is true for a majority of $\mathbf{I}$, then only a fraction of the actual state space contributes to the solution and needs to be explored, making a brute-force approach feasible.

We would just have to note the largest set for which a given move is optimal; this move would be optimal for its subsets, as well, except those explicitly listed in a separate entry. This reasoning makes the trivial assumption that reducing the number of states, and thus uncertainty, cannot worsen the worst-case performance of any move; we can simply ignore the additional information.

Roughly speaking, the algorithm proceeds through iterative retrograde analysis and creates a table of entries of which the elements contain an information set or metaposition, an optimal move associated with it, and a maximum number of moves this strategy will take to achieve certain victory. First, we find all metapositions $i_1 \in \mathbf{I}$ such that checkmate is possible in one move. Figure 1, (c) shows one of the only examples of single-move checkmates in KRK with more than one king location. These mates are very simple to find; it suffices to search all piece configurations and the corresponding legal moves for checkmate positions according to chess rules.

After solving the problem at depth 1 as described above, we look for all $i_2 \in \mathbf{I}$ such that checkmate is possible in at most two moves no matter what the referee's response is; this includes situations always leading to subsets of states found in the first step. This procedure must be repeated several times at each depth until no new entries are generated; we need to do this in order to deal with illegal moves which require analysis of same-depth metapositions. When the algorithm performs a full run through all the existing positions without adding anything to the tablebase, the current depth is exhausted and we proceed to the next depth. We also ignore metapositions that are subsets of previously added metapositions, since they provide no new information. When the algorithm fails to generate any new entries for the next depth, execution ends.

The algorithm is summarized in Fig. 2. The basic concept is that, if we know that, no matter the referee's response message, we are going to mate the King in at most n moves, then the present position is won in at most $n+1$ moves. If the method by which a new position is constructed is correct, then the correctness of the whole algorithm is easily proved by induction. One needs to be careful in how $(n+1)$-depth entries are constructed from the n-depth (and below) entries, because the danger of "strategy fusion", as defined by Frank and Basin [15], is always around the corner. Strategy fusion can be briefly summarized as the pitfall of having plans for dealing with each specific case successfully, but not knowing which of those cases we are in. Within the context of Kriegspiel endings, this happens if we mistakenly assume that, only because we can solve metapositions $i_1, i_2 \in \mathbf{I}$ in n moves, we can also solve $i_1 \cup i_2$ in n moves. This is not true even if the optimal strategies for i_1 and i_2 start with the same move; there is no guarantee that the same move will also solve $i_1 \cup i_2$ optimally, or that it can be solved at all. Such a pitfall is showcased in the KPK ending in [4], where victory cannot be achieved with probability 1 under certain circumstances.

Therefore, the white player needs to know, at any given time, what metaposition he is in. This is accomplished by the innermost *for* loop, where all compatible metaposition entries in the current list (that is, all entries with the correct placement of white pieces) are assigned to the possible referee messages

```
kriegRetrograde(entryList,depth)
begin
 added = false;
 for each placement of white pieces P do
  fill P with black kings
  for each pseudolegal move M do
   messages = possibleMessages(P,M);
   for each assignment A of entries from entryList to messages do
    reduceBlackKings(P,A);
    if (checkAndAdd(entryList,P,A)) added = true;
   od
  od
 od
 if (added) kriegRetrograde(entryList,depth);
 else if (entriesWithDepth(depth+1)) kriegRetrograde(entryList,depth+1);
 return entryList;
end
```

Fig. 2. Pseudocode listing for main retrograde function

that may result from the move being examined. In the least optimized case, all possible combinations might be tested. For example, if move Ka2-b2 can generate a check, a silent response and an illegal notice, the algorithm will generate $n_1^2 \cdot n_2$ assignments, where n_1 is the number of metapositions already in the database with the white pieces placed just like the current board after Ka2-b2 (squared because there are two messages to consider), and n_2 is the number of metapositions where the white pieces are set up like the current board (for an illegal outcome). For each assignment of metapositions $i_1, \ldots, i_k$ to messages $m_1, \ldots, m_k$, the algorithm determines the largest metaposition such that every message m_j will result in a subset of the corresponding metaposition i_j. Since every possible outcome has already been computed as won, this new metaposition is also won. Since all assignments are considered, optimal assignments will also be found, and sub-optimal assignments will be discarded.

Method `reduceBlackKings` generates new metapositions in a subtractive way, starting with a full board and taking away black Kings as they are found to be incompatible with a message-outcome pair; this means that the algorithm decides what message m_j would be generated if the black King were there, and if the King's positions after Black's next move do not entirely lie in i_j, that square has to be cleared. If the resulting metaposition is empty, it is discarded. If it is a subset of an existing entry, it is also discarded. In contrast, if the new metaposition is original, it is added to the entry list together with the tested move and depth information. If a new entry is added, the algorithm will have to go through an additional pass; illegal moves may generate new mates of depth n instead of $n + 1$, which need to be searched again. Most depth levels in KRK require 3 to 5 passes, after which the next depth is considered.

Once the algorithm has finished, it returns a full list of metapositions, each having a best move and a distance to mate in the worst case. This database is used as follows. The player searches it with a metaposition representing the current state of the game. All entries that are supersets of it are returned, and if none exist, it means that it is not possible to force a mate from here. Among these entries, the player selects the one with the shortest distance to mate; in the event of a tie, he will pick the one with the lowest number of states (black Kings). He will then proceed to play the corresponding move.

3.1 Complexity and Optimizations

The computational complexity of a single depth step of the algorithm is $O(b^p n^m)$, where b is board size, p is the number of white pieces on the board, n is the average size of the metaposition list, and m is the number of messages the referee can output. Estimating n is best done through the actual experiment, which shows that its increase in KRK is exponential and quite regular until about depth 30, after which there is a sharp slowdown. The algorithm can become very slow if there are many possible referee messages. This will be the main problem to overcome with the KQK ending, since the Queen can check in four different ways, and most moves can result in three (as opposed to just one in KRK): file, rank, short diagonal and long diagonal.

The following optimizations are possible and have been implemented.

– Game-ending referee messages are handled implicitly. Black King positions leading to instant checkmate are automatically added to any new entry, and stalemate or drawn positions are excluded before anything else takes place. This means that two referee messages can be taken off the list, leaving only three in KRK - silent, illegal, and check.
– Not every metaposition assignment is considered, but only those that will create a metaposition of the desired depth. For example, if we are filling the database with depth 10 entries, it is useless to try an assigment leading to depth 9 or less, because it will already have been considered at the appropriate step. In other words, in order to generate a depth 10 entry, we need to have at least one depth 9 entry assigned to the silent or check messages *or* another depth 10 entry linked to the illegal move message. Any assignment violating this rule is skipped.
– Not every single metaposition in the list is checked. In particular, at the end of each step, metapositions that are found to be subsets of others are "dumped" into support files, regardless of their distances to victory; the only exception is metapositions generated during the last step. This is related to the above point; when determining depth 10 entries, there will always be at least one depth 9 or depth 10 metaposition in the assignment, but as for the others it is irrelevant whether their depth is 8 or 2. One can simply take the larger one and dump the smaller. As a consequence, the final database is obtained by merging all the dump files with the collection that remains after the last iteration of the algorithm.

– A further optimization to reduce the number of metapositions being considered is to perform an intersection between each entry and the legal positions of the black King after the corresponding message. In this way, only the relevant parts of the metaposition are considered, and duplicates can be computed only once.

4 Solving KRK

We ran the algorithm described in the previous section in order to ascertain whether it is computationally feasible for the KRK ending, and if so, whether existing directives such as Boyce's and Magari's can be improved on. Finding a measure of the complexity for this problem (that is, how many metapositions it takes to describe KRK completely out of the theoretical 10^{16}) and the longest forced KRK mate were also goals for this computation. We implemented the algorithm in the Java programming language and executed it on a single machine. The algorithm did not feature any peculiar optimizations besides those described in the previous section, and it ran on a single processor. Its run terminated after about 12 days of uninterrupted execution.

Using mirroring on the x, y, and diagonal axes, the problem of KRK in Kriegspiel is described with a tablebase of 1,087,599 metapositions, or about 10^6 from the hypothetical 10^{16}. KRK in chess is fully described with about 23,000 positions, making the equivalent Kriegspiel problem about 40 to 50 times as complex. Results may vary to a degree, depending on storage policies for metapositions that are subsets of other entries; in particular, the tablebase can be compressed to unify a subset with its superset if they have the same best move, but doing so loses information on the strict upper bound on the moves till victory, which has to be recalculated on the fly with more accesses to the tablebase.

The longest forced mate sequences in Kriegspiel KRK are 41 moves long, making the 50-move rule irrelevant in this endgame, and there are only two. One of them is shown in Fig. 3, (a). The other mate in 41 is almost identical, except that the Rook is on h4 instead of g4; the solution is the same. It is readily seen that such a position cannot occur during a normal game. The same can be said for a majority of entries in the tablebase, especially at the later depth levels. After depth 35, most entries resemble man-made puzzles and problems rather than situations that a player is likely to encounter. In this particular problem, the player must maneuver his pieces around the enemy Kings until he safely brings the Rook next to the King. In the optimal solution, it takes nine moves to accomplish this: Rf4, Kc2, Rf8, Kd3, Rg8, Rh8, Rh1, Rd1, Kc2.

Boards (b) and (c) represent situations that are much more likely to happen in a real game. In particular, board (b) is the starting position for Boyce's algorithm given in [9], and board (c) is the starting position for Magari's algorithm, from [7]. Boyce's directives are based on trapping the King in a single quadrant of the board with the rook and then using the King to push back the opponent. Magari's method consists of starting from (c) and isolating the King on one side

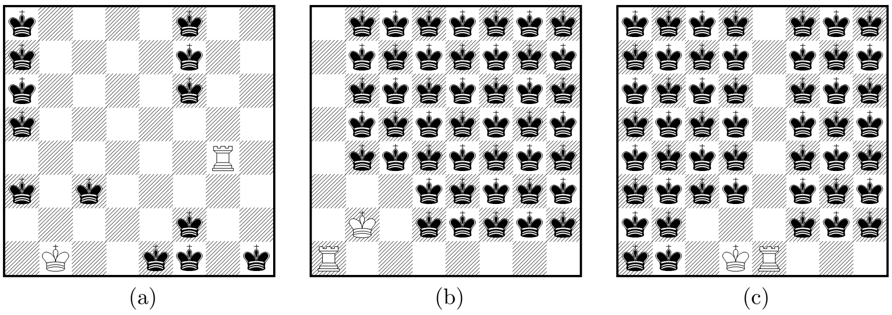


Fig. 3. (a) Longest forced mate in KRK, mate in 41; (b) Boyce's starting position, mate in 30; (c) Magari's starting position, mate in 34

of the board by playing Kd2, Re2, Kd3, Re3, and so on, scanning the board until a check reveals the location of the enemy king. The tablebase shows that Boyce's method is a better approximation of the shortest mate. Boyce's position is a mate in 30, four moves shorter than Magari's position. There are only two positions completely filled with enemy Kings that can be won faster, in 29 moves, and both have the Rook in the a1 corner, just like (b); the King is on c2 and c3, respectively. Thus, Boyce has a good understanding of a convenient starting position. Moreover, optimal play from (c) does not follow Magari's algorithm. Instead, the player immediately runs to the nearest corner, reaching a Boyce-like position through, for example, Ke2, Kf2, Rg1, Kg3, Rh1 - from here White mates in 29, as mentioned. A full run of the algorithm still reminds of Boyce's directives, though it is noticeably more quirky and difficult to summarize.

5 Conclusions and Future Work

We have established an algorithm that will always win the KRK endgame if victory can be achieved with probability 1, and instantly provide a strict upper bound on the number of moves until checkmate. This approach can be extended to other Kriegspiel endgames such as KQK, KPK, and others. Currently, the KRK tablebase occupies about 80 megabytes of hard disk space, and a different structure might be needed in order to deploy it into Kriegspiel-playing programs, as looking up the best move in a given situation is not as straightforward as in chess: on average, the program has to examine 25,000 metapositions and find the compatible candidate with the shortest route to mate. Extension to queen endgames will also probably require more optimizations, as preliminary tests have shown an almost tenfold decrease in speed due to the exponential nature of the search algorithm.

Another area of improvement would involve choosing among equivalent lines of play (in terms of worst-case performance) by reasoning and making assumptions on the opponent. This feature could be helpful in shortening mate sequences if

Black does not play near-optimal moves. It is to be noted that playing almost like an oracle is not too difficult in this endgame; a King trying to maintain position by moving back and forth near the center of the chessboard is often playing the best strategy, or close to the best.

References

1. Zermelo, E.: Uber eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In: Proceedings of the Fifth International Congress of Mathematicians, Cambridge, UK, vol. 2, pp. 501–504 (1913)
2. Stiller, L.: Some Results from a Massively Parallel Retrograde Analysis. ICCA Journal 14(3), 129–134 (1991)
3. Bellman, R.: On the application of dynamic programing to the determination of optimal play in chess and checkers. Proceedings of the National Academy of Sciences 53(2), 244–247 (1965)
4. Ciancarini, P., DallaLibera, F., Maran, F.: Decision making under uncertainty: a rational approach to Kriegspiel. In: van den Herik, J., Uiterwijk, J. (eds.) Advances in Computer Chess, vol. 8, pp. 277–298. Univ. of Rulimburg (1997)
5. Ferguson, T.: Mate with bishop and knight in Kriegspiel. Theoretical Computer Science 96, 389–403 (1992)
6. Ferguson, T.: Mate with two bishops in Kriegspiel. Technical report, UCLA (1995)
7. Magari, R.: Scacchi e probabilità. In: Atti del Convegno: Matematica e scacchi. L'uso del Gioco degli Scacchi nella didattica della Matematica, Forlì, Italy, pp. 59–66 (1992)
8. Sakuta, M.: Deterministic Solving of Problems with Uncertainty. PhD thesis, Shizuoka University, Japan (2001)
9. Boyce, J.: A Kriegspiel endgame. In: Klarner, D. (ed.) The Mathematical Gardner, pp. 28–36. Prindle, Weber & Smith (1981)
10. Bolognesi, A., Ciancarini, P.: Searching over metapositions in Kriegspiel. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 246–261. Springer, Heidelberg (2006)
11. Russell, S., Wolfe, J.: Efficient belief-state AND-OR search, with application to Kriegspiel. In: Int. Joint Conf. on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, pp. 278–285 (2005)
12. Nance, M., Vogel, A., Amir, E.: Reasoning about partially observed actions. In: Proceedings of 21st National Conference on Artificial Intelligence (AAAI 2006), Boston, USA, pp. 888–893 (2006)
13. Ciancarini, P., Favini, G.: Representing Kriegspiel states with metapositions. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), Hyderabad, India, pp. 2450–2455 (2007)
14. Parker, A., Nau, D., Subrahmanian, V.: Game-tree search with combinatorially large belief states. In: Int. Joint Conf. on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, pp. 254–259 (2005)
15. Frank, I., Basin, D.: A theoretical and empirical investigation of search in imperfect information games. Theoretical Computer Science 252, 217–256 (2001)

Conflict Resolution of Chinese Chess Endgame Knowledge Base

Bo-Nian Chen[1], Pangfang Liu[1], Shun-Chin Hsu[2], and Tsan-sheng Hsu[3,⋆]

[1] Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan
{r92025,pangfeng}@csie.ntu.edu.tw

[2] Department of Information Management, Chang Jung Christian University,
Tainan, Taiwan
schsu@mail.cjcu.edu.tw

[3] Institute of Information Science, Academia Sinica, Taipei, Taiwan
tshsu@iis.sinica.edu.tw

Abstract. Endgame heuristics are often incorperated as part of the evaluation function used in Chinese Chess programs. In our program, CONTEMPLATION, we have proposed an automatic strategy to construct a large set of endgame heuristics. In this paper, we propose a conflict resolution strategy to eliminate the conflicts among the constructed heuristic databases, which is called *endgame knowledge base*. In our experiment, the correctness of the obtained constructed endgame knowledge base is sufficiently high for practical usage.

1 Introduction

A game of Chinese Chess, like chess, can be divided into three phases: (1) opening game, (2) middle game, and (3) endgame. The opening game is the first phase of a game in which almost all pieces are on the board. After about 20 plies, both players have moved their pieces to important places and the game turns into the middle-game. After exchanging some pieces, the game goes into the endgame phase.

The most popular technique used to solve the opening-game problem is constructing opening databases that store all possible choices from previous games [1]. In the middle game, people often use a nega-scout algorithm with a good evaluation function and an appropriate move-ordering scheme to obtain a good solution [2]. There are also strategies in artificial intelligence that automatically generate middle-game evaluation functions [3]. In the endgame phase, the performance of today's Chinese Chess program is still not satisfiable compared to human experts. People often solve relatively small endgames by using retrograde algorithms [4].

Endgame databases constructed by retrograde analysis algorithms are perfect in the sense that the game-theoretical values of all positions matched are available in the database . However, there are two drawbacks in endgame databases:

⋆ Corresponding author.

(1) they need too much memory, and (2) practical endgames contain too many pieces for current retrograde algorithms to handle. Hence, the search algorithm still needs heuristic information about endgames. The challenge here is: how to generate effective heuristics for the endgames in question? Although heuristics are not perfect, they can be applied to practical endgames and are useful in tournaments.

Our intuition is as follows. Each endgame is assigned with a heuristic, namely a level of [advantage, disadvantage] according to the material combination of the two sides. This assigned value reflects the heuristic value that a Chinese Chess master usually assigns to the endgame, since he[1] understands whether the endgame is advantageous or not without considering the positions of the pieces. The heuristics of two endgames may be obtained using different methods, such as from text books or human annotation. These heuristics, though they have a high level of accuracy, may still contain errors. We have observed that if the material combinations (a definition is in 2.1) of two endgames differ by a small number of pieces, then the heuristic values of the two endgames are not independent. For example, knowing that an endgame usually is advantageous to the Red side, then the new endgame heuristic value after adding a Red piece cannot be worse than the original one. In general, assume that an endgame A is related to many other endgames and further assume that a large portion of the heuristics obtained so far is correct, then A must be consistent with most of the related endgames. If this is not the case, then there is a high chance that the heuristic value of A is incorrect. Using this high level idea, we build an expert system to self-correct a large set of annotated heuristics.

In our previous work, we have designed a method to generate automatically a large number of heuristics of material configurations [5]. However, we have detected some conflicts in our endgame knowledge base. A small number of conflicts are sufficient to harm our search algorithms. Hence, we propose an adequate conflict reduction algorithm that increases the consistency in the endgame knowledge base.

In this paper, we will discuss the following important issue relative to the endgame problems: what is the impact of a piece exchange when transforming the game from a middle game to an endgame? In Chinese Chess, using solely material values computed from summing all piece values to choose good exchanges may be incorrect and that only in some cases. Our solution is to use the many heuristics of the material combinations, called the endgame knowledge base, to guide our program when piece exchange is needed.

2 Theoretical Foundations

In this section, we describe the theoretical concepts about lattices as used by our method and discuss the problems that occur in our automatic generated endgame knowledge base.

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

2.1 Using Lattices to Represent the Material Structure

There are seven types of pieces in Chinese Chess: king (K), guard (G), minister (M), rook (R), knight (N), cannon (C), and pawn (P). A *material combination* is defined as the set of pieces in a position, e.g., KCMKRP is a material combination that the red player has the king, a cannon and a minister; the black player has the king, a rook, and a pawn. To a material combination a score is attached that describes its advantage without position information. Each material combination has exactly one *mirrored material combination* such that the red pieces and the black pieces are swapped. The *material structure* consists of a set of material combinations. In our discussion, the material structure represents all of the material combinations in our endgame knowledge base. *Invariable nodes* are those modified or verified by a human expert. They cannot be changed by our conflict reduction algorithm.

The material-combination structure can be viewed as a lattice. A lattice is a partially ordered set(poset) in which all non-empty subsets have a *join* and a *meet*, as defined in mathematical order theory. A join is the *least upper bound* of an element or a subset; a meet is the *greatest lower bound* of an element or a subset. All material combinations in Chinese Chess follow the *piece additive rule* that for a material combination, adding pieces to a player cannot make him be disadvantageous if we only consider the material combination, not the specific positions. The piece additive rule also claims that removing a piece from a material combination cannot be better than the original material combination. By applying the piece additive rule, the material structure can be transformed into a lattice which is a directed graph. The node in the lattice represents a material combination. The edge connects two material combinations that differ only in one piece. We define $x \rightarrow y$ as two adjacent nodes and the directed edge represents that the red player is at least as advantageous in x as in y. In lattice, $x \rightarrow y$ means that x and y are comparable and the meet of x is y.

2.2 Construction Strategy of Material Structure

In the lattice, the least element is KK. We always expand the material structure by adding either a red piece or a black piece. The number of possible piece types on one side is $3^5 \times 6 = 1,458$. Totally, there are $1,458^2 = 2,125,764$ possible material combinations in Chinese Chess. An example of material structure is shown in Fig. 1. A lattice can be divided into several levels. Each level contains material combinations of the same piece number. In Chinese Chess, there are at most 32 pieces and at least 2 pieces, two kings, in a position. Hence, there are totally 31 possible levels in our lattice.

It is not always correct to give a score to the material combinations on each side. For example, KPPKGG and KPPKMM are generally red-win endgames when pawns are not yet moved to the palace of the opposite side in the starting position of these endgames, but KPPKGM is generally a draw endgame. Thus, we may conclude that KGM is better than KGG and KMM for defense. However, KNPKGM and KNPKGG are generally red-win endgames but KNPKMM is a

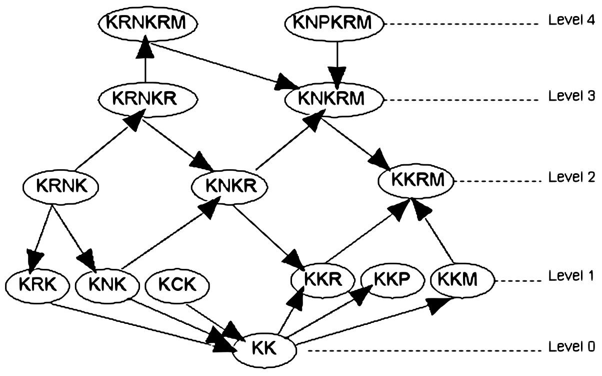


Fig. 1. An example of lattice structure for some endgames

draw endgame when the pawn stands in the last line of the palace of the opposite side. The conclusion that KMM is better than KGG and KGM is inconsistent with the last case.

The score value of a node is in the range of [0 − 9]. The values 0 (win), 1 (most win), 2 (advantage), 3 (slight advantage) represents the score that the red side is in advantage, 4 represents that any one player has a chance to win, 5 means almost draw, 6 is oppsite to 3, 7 is opposite to 2, 8 is opposite to 1, and 9 is opposite to 0. In our automatic endgame knowledge base construction, we first create a basic endgame knowledge base manually. A probablistic method is used to evaluate the score value of the generated material combinations. By using a systematic generating algorithm, we have constructed a large endgame knowledge base. The endgame knowledge base is used by a middle-game search algorithm to find paths to enter advantageous endgames by exchanging pieces.

Inconsistency is defined as the case that score values of adjacent nodes which violate the piece additive rule. The two nodes are called *inconsistent nodes*, the corresponding edge is called *inconsistent edge* and the corresponding connected node is called the *Inconsistent neighbor*. *Inconsistent percentage* of a node indicates the number of inconsistent neighbors divided by the number of its neighbors.

Although most of the score values in our endgame knowledge base are accurate, only a little conflict is sufficient to let the search algorithm select wrong moves. To resolve the problem, we propose a conflict reduction algorithm that reduces the number of inconsistent nodes. When our algorithm cannot progress further, we ask the help of a Chinese chess expert to do a small amount of modification. Then we rerun our self-correcting algorithm. After some iterations of modification, we can obtain a zero-conflict endgame knowledge base. To further increase correctness, we ask our Chinese chess expert to verify a randomly selected sample to evaluate the percentage of errors and propose further modifications.

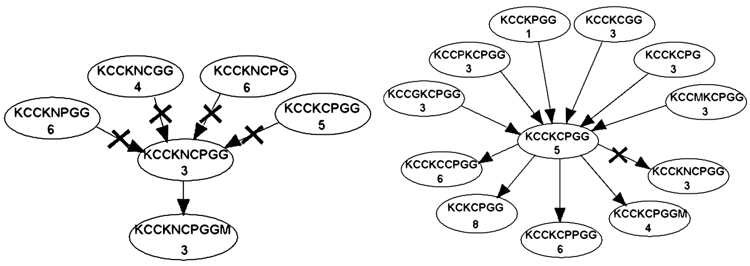


Fig. 2. Examples of two adjacent inconsistent nodes. The number in each node is its score value. The edge with a cross means an inconsistent edge.

3 Basic Conflict Reduction Algorithms

In our automatic-generated endgame knowledge base [5], there are some inconsistent material combinations that causes the search algorithm to exchange a piece incorrectly. For example, consider two adjacent material combinations, as shown in Fig. 2. To illustrate the concept of conflicts, we can find that the central node, KCCKNCPGG, has a score value 3, and the leftmost node, KCCKNPGG, has a score value 6. There is a conflict because the black side is more advantageous when a black cannon is taken. The first material combination has four inconsistent edges, so the inconsistent percentage is 80%. The second material combination has only one inconsistent edge, and the inconsistent percentage is 9.09%. In this example, the first material combination is more likely to be incorrect and should be modified first.

We need a standard endgame knowledge base that is considered as "correct" for the conflict reduction algorithm. The algorithm discovers and modifies the nodes in the automatically generated endgame knowledge base that are inconsistent with some nodes in the standard endgame knowledge base.

3.1 Conflict Computation

If there is a conflict in a lattice, there must be some nodes having inconsistent neighbors. A conflict computation procedure computes the number of inconsistent neighbors of each nodes. When finishing the computation, information that we actually want to know is (1) the inconsistent number and (2) information of the inconsistency level. The level of inconsistency is relative to the inconsistent percentage of the node. We define 10 levels of inconsistency, from level 0, level 1, ... , to level 9. Level 0 represents the inconsistent percentage in (0% – 10%], level 1 represents (10% – 20%], etc. Information of the number of nodes is in each inconsistency level. It simplifies the process of reducing conflicts.

Our idea is a greedy method that always modifies a node of the highest inconsistency level. We use *inconsistent edge checking* to find conflicts between nodes. An inconsistent edge-checking algorithm has two targets: (1) two neighbor nodes, and (2) two mirrored material combinations. We implemented the piece additive rule which is defined in Section 2.1 for the first target. For the second target, a mirrored material combination pair in our lattice is virtually considered as the same material combination. An inconsistent edge-checking algorithm can ensure the consistency of mirrored material combinations.

The algorithm of computing conflicts straightforwardly uses inconsistent edge checking to summary the inconsistency neighbors for each node. Computing conflicts of the whole lattice can be performed in $O(MN)$ time, where M is the maximum number of the neighbors of a node, and N is the number of nodes.

3.2 Conflict Reduction Algorithm

The conflict reduction algorithm, shown in Algorithm 1, finds inconsistent nodes in our lattice and modifies the score values of some nodes to reduce the number of inconsistent nodes. It takes four steps to finish the work: (1) conflict computation, (2) candidate selection, (3) score value selection, and (4) modification. Our algorithm repeats the four steps until no more candidate can be selected. The first step, conflict computation, is described in Section 3.1. The next three steps are described as follows.

```
procedure ConflictReduction()
    do loop
        err_num = ConflictComputation();
        ResetUpdated(); // set the updated flag as not updated
        if(err_num = 0 or no any modification)
            break;
        do loop
            node = CandidateSelection();
            if(no valid node)
                break;
            score = ScoreValueSelection(node);
            Modify(node, score); // change score value
        end loop
    end loop
end procedure
```

Algorithm 1. Conflict reduction algorithm

The first step, candidate selection, chooses the node with the highest conflict rank to be modified. Score value selection then tries all possible values and computes the inconsistent percentage after modification. The value with the minimum inconsistent percentage is selected. If two scores of a node have the same minimum inconsistent percentage, we ask the help of our human expert. A detailed discussion takes place in Section 4.3. During modification, we need to update simultaneously the mirrored material combinations when modifying a node to ensure their consistency.

In our algorithm, an updated flag is used to avoid repeat selection of the same candidate in one iteration. An invariable flag identifies whether a node is modified by our 4-Dan expert and should be removed from the candidate selection procedure.

4 Refinements

In this section, we introduce the diffusing algorithm (4.1) and other enhancing techniques (4.2). We also discuss the verification issue of a consistent lattice (4.3).

4.1 Diffusing Algorithm

A diffusing algorithm, as shown in Algorithm 2, is a recursive procedure that searches the neighbor nodes for nodes with only one *consistent possible value*, which is the value that is not inconsistent with any invariable nodes. These nodes are trivial and should be modified first.

By executing the diffusing algorithm, all nodes with only one possible value are modified first. In this way our algorithm attempts to reduce more conflicts.

```
procedure Diffusing(n)
    // variable n represents the node to be diffused
    // variable nn represents a neighbor node of variable n
    if(n is updated before)
        return;
    for each nn of n do
        (count, score) = ComputeRelativePossibleValue(nn, n);
        if(count = 1)
            Modify(nn, score);
            Diffusing(nn);
        end if
    end for
end procedure
```

Algorithm 2. Diffusing algorithm

4.2 Ranking and Scoring Strategies

Subsequently, we rank the nodes in the lattice to indicate its degree of errors. This is called the *conflict rank*. Our algorithm takes an element with the highest rank to update its value. In the basic conflict-reduction algorithm, we use an inconsistent percentage as its conflict rank. Here we define a better conflict rank:

$$V = N_i \times N_n.$$

In the above formula, V represents conflict rank, N_i represents the number of inconsistent neighbors, and N_n represents the number of neighbors. Instead of dividing by N_n, the new conflict rank is multiplied by N_n to emphasize the importance of the number of neighbors.

In the step of score-value selection, we use the notion *corrected score* to measure the whole level of inconsistency. Our conflict-reduction algorithm always selects the score value that minimizes the correct score. In our basic method, we use the number of inconsistent nodes as a correct score. So, we have developed a new correct score that favors small inconsistency levels as follows:

$$V_c = \sum_{i=0}^{9} 2^i \times I_i.$$

In the formula, the value V_c means corrected score, I_i represents the information of inconsistency level i (see Section 3.1).

By using the new corrected scores, nodes with large inconsistent percentages are usually reduced to smaller percentages. The new corrected score improves the ability of identifying better score values and thus decreases the probability of fall into local minimum.

4.3 Final Verification

Now assume that we have obtained a consistent endgame knowledge base. Then, still, there may be two types of errors in the lattice: (1) an fully isolated subset of the lattice is incorrect in all its elements; and (2) there are errors in some nodes that do not influence the consistency.

Checking by random sampling verification is a way to obtain the approximation of the correctness of the lattice. For this purpose, we select a small number of nodes with a percentage p in the lattice randomly such that the distance of any two selected nodes is at least k. The selected material combinations are verified by a human expert. If n error nodes are reported, the approximated value of the whole set of error nodes is n/p. The modified data can also be used to reduce the errors of the whole knowledge base.

5 Experimental Results

In this section, we use the practical endgame knowledge base in CONTEMPLATION as our test data (5.1). Then (in 5.2) we present (1) the reduction ability of the conflict reduction algorithms, (2) the correctness analysis by random sampling verification, and (3) the comparison of the consistent endgame knowledge base with its original version.

5.1 Experimental Design

Our test data is the endgame knowledge base used by our program, CONTEMPLATION. There are three manually constructed endgame knowledge bases: END65, END60, and END50. The number of nodes in END65, END60, and END50 are

17,038, 422, and 1,499, respectively. The data to be tested is in our automatic generated endgame knowledge base, i.e., END64, END59, and END49 defined by methods as used in [5]. An extended endgame knowledge base is identified by the number of the original knowledge base decreased by 1, i.e., END64, which is generated by extending END65, contains the neighbors of the nodes in END65. The number of nodes in extended knowledge bases END64, END59, and END49 are 47,621, 2,722, and 3,938, respectively. Our practical endgame knowledge base, ENDALL, combines six endgame knowledge bases, containing 69,595 nodes.

5.2 Results and Discussions

In the first experiment, we tried our methods on four endgame knowledge bases, viz. END64, END59, END49, and ENDALL. There are two methods which we would like to be compared: (1) basic conflict reduction algorithm (called BA), and (2) the algorithm with all refinements (called RA). The results are shown in Table 1. The number of iterations indicates the iterations needed for convergence. An iteration means handling all inconsistent nodes once in the lattice. To test the convergency, we need an extra iteration to ensure that no modifications are performed in an iteration.

Table 1. Comparison of the reduction ability of the basic algorithm and the refined algorithm. The "error after BA" and the "error after RA" columns show the number of inconsistent nodes after performing the basic algorithm and the refined algorithm, respectively.

	DB size	org error	error after BA	iterations	error after RA	iterations
END64	47,621	14,616	9,786	6	970	3
END59	2,722	1,786	1,330	3	166	2
END49	3,938	1,362	438	6	45	3
ENDALL	69,595	16,488	11,108	6	585	4

By using all refinement techniques, we obtain knowledge bases with much less conflicts in less number of iterations than previously was the case. It reduces the work of a human expert to verify and modify the endgame knowledge base considerably.

In the second experiment, we prove the correctness of our endgame knowledge base. We do so after we performed random sampling verification. We executed three random sampling experiments. In each experiment, we used an algorithm described in Section 4.3 to generate 695 different nodes with parameters $k = 4$, and $p = 1\%$. After the sampled nodes have been verified and modified, we performed our conflict reduction algorithm with all refinement techniques on the ENDALL knowledge base. Here we define the *error distance* as the difference between the score value of the consistent endgame knowledge base and the score value verified by our human expert. Because the score values 4 and 5 represent very similar classes of advantage, the error distance between them is set to zero. In addition, the error distance between any score value and 4 is considered equal to the error distance between that score value and 5. We recorded the error distances of the modified data and checked whether it was inverted. An inverted result is a result that is wrong on the side who has the advantage. For example, a node that represents that the red side has an advantage and is marked as a black win is an inverted result. The result of the full experiment is shown in Table 2.

Table 2. The statistical analysis of the correctness in the zero-conflict endgame knowledge base ENDALL. IR means inverted results. D represents error distance.

	ErrNum	$D \geq 4$	$D = 3$	$D = 2$	$D = 1$	$D = 0$	IR
Sample1	92	0	2	15	75	0	1
Sample2	127	0	2	9	116	0	1
Sample3	99	0	2	16	80	1	0
Average	106.0	0.0	2.0	13.33	90.33	0.33	0.66
n/P	10600	0	200	1333	9033	33	66
%	15.23	0	0.28	1.91	12.97	0.00	0.00
Confidence	1533	0	200	1333	0	0	66
%	2.20	0	0.28	1.91	0.00	0.00	0.00

The probability of having an error distance of more than one is 2.20%. Note that a node with a score value 4 or 5 cannot be inverted. Hence, inverted results only happen when score values are less than 4 or more than 5. In other words, inverted results happen when the distance is more than or equal to one. They are also counted when the column of the distance is more than or equal to one. Although the absolute correctness is 85.77%, which is acceptable, the correctness with confidence is 97.70% ignoring one level difference. Evaluation of a material combination as "win" or "win in most cases" is a subjective choice. Even a human master or grandmaster may have a subjective judgement in many practical positions. Hence, we assume that a difference of 1 level is tolerable.

In Table 3, we show (1) the statistical comparison of the consistent endgame knowledge bases after verification and (2) their original versions. Since different original endgame knowledge bases may contain identical material combinations, we have filtered them when merging endgame knowledge bases. The score values of identical material combinations is set as the material combinations that first appear in the merging process. Hence, some material combinations may be counted more than once. They may even contain different original score values. For example, in END65, there are two errors with zero error distance, which are KNNMKNNG and KNNGKNNM. There are also two errors with zero error distance in END59, which are KCCGKCC and KCCKCCG. During the merging operation, they are set as correct score values by chance, and the number of errors with zero error distance in ENDALL becomes two.

In total, we have modified 24,486 material combinations in the final consistent endgame knowledge base. In the original endgame knowledge base, there are 1,652 errors of a distance more than or equal to four, 2,392 errors of a distance three, 2,286 errors of a distance two, 18,154 errors of a distance one, two errors between a score value 4 and 5, and 1,064 inverted results.

About 9.10% of the original ENDALL knowledge base contain errors of $D \geq 2$; about 26.09% of them contain errors of $D = 1$; there are no errors of $D = 0$; about 1.53% of them contain inverted results.

Table 3. The statistical comparison of the original endgame knowledge bases and the final version of the consistent endgame knowledge bases. IR means inverted results. D represents error distance.

	DB size	ErrNum	$D \geq 4$	$D = 3$	$D = 2$	$D = 1$	$D = 0$	IR
END65	17,038	1,100	6	24	174	894	2	80
END60	422	48	2	6	8	32	0	8
END50	1,499	222	4	16	34	168	0	10
END64	47,621	20,908	1,056	2,042	1,952	15,858	0	708
END59	2,722	1,734	594	390	142	606	2	290
END49	3,938	1,982	60	32	50	1,840	0	50
ENDALL	69,595	24,486	1,652	2,392	2,286	18,154	2	1,064

6 Conclusions and Future Work

A complete mastering of the Chinese Chess endgame problem is a hard problem, even today. We constructed an endgame knowledge base for our search algorithm to identify which kinds of endgames are beneficial. In this paper, we propose a conflict reduction algorithm to resolve the conflicts in our automatically generated endgame knowledge base. The strategy is effective when handling large knowledge bases with a relatively small percentage of conflicts. The resulting endgame knowledge base obtained is checked by a random sampling verification and received high accuracy. We used this modified knowledge base in our program, Contemplation, and found it to improve steadily its strength against the previous version. Its correctness is sufficiently high for practical usage.

In the future, we will enhance our conflict-reduction algorithm to be more sensible of advantage. For example, KCCKCPGG and KCCKCGG differ only by one Pawn and they have score values 5 and 8, respectively. KCCKCRGG and KCCKCGG differ by a Rook and they also have score values 5 and 8. The degrees of conflict-free expectation of the two above cases are different. In practical usage, we see the following. If the two cases are both inconsistent, then the latter case has a more severe degree of conflict than the former case. A second example is KRPKGGMM and KRPKGGM that are assigned 0 and 9 respectively when compared with the same material combinations who are assigned 0 and 1, respectively. Although two cases include a conflict, the first score values are more severe because the difference of the score values is quite obvious.

Combining the two representative examples, we can define a lattice with weighted edges. The weight is defined as follows:

$$w = D_m(m_1, m_2) \times D_s(Score(m_1), Score(m_2))$$

The variable w represents the weight which indicates the degree of conflict. Function D_m computes the difference between the two material combinations m_1, and m_2; function D_s computes the difference between the two score values of m_1, and m_2. The weight value follows the order: $rook > cannon = knight > pawn > guard = minister$. The weight of guards and ministers should be adjusted dynamically: when the player has cannons, the weight values of guards and ministers should be bigger than the ones without. A function score retrieves the score value of a material combination. When a conflict occurs, the node with a larger weight value first needs to be taken care of.

Acknowledgment

The authors Hsu, T.-s., Chen, B.-n. are supported in part by National Science Council (Taiwan) Grants 97-2221-E-001-011-MY3.

References

1. Chen, J.-c., Hsu, S.-c.: Construction of online query system of opening database in computer Chinese chess. In: The 11th Conference on Artificial Intelligence and Applications (2001)
2. Yen, S.-j., Chen, J.-c., Yang, T.-n., Hsu, S.-c.: Computer Chinese chess. ICGA Journal 27(1), 3–18 (2004)
3. Chen, B.-n., Liu, P.-f., Hsu, S.-c., Hsu, T.-s.: Abstracting knowledge from annotated Chinese chess game records. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 100–111. Springer, Heidelberg (2007)
4. Wu, P.-s., Liu, P.-y., Hsu, T.-s.: An external-memory retrograde analysis algorithm. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 145–160. Springer, Heidelberg (2006)
5. Chen, B.-n., Liu, p., Hsu, S.-c., Hsu, T.-s.: Knowledge inferencing on Chinese chess endgames. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 180–191. Springer, Heidelberg (2008)

On Drawn K-In-A-Row Games

Sheng-Hao Chiang[1], I-Chen Wu[2], and Ping-Hung Lin[2]

[1] National Experimental High School at Hsinchu Science Park, Hsinchu, Taiwan
jiang555@ms37.hinet.net
[2] Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{icwu,bhlin}@csie.nctu.edu.tw

Abstract. In 2005, Wu and Huang [9] presented a generalized family of *k*-in-a-row games. The current paper simplifies the family to *Connect*(k, p). Two players alternately place p stones on empty *squares* of an *infinite* board in each turn. The player who first obtains k consecutive stones of *his*[1] own horizontally, vertically, diagonally wins. A *Connect*(k, p) game is drawn if both have no winning strategy. Given p, this paper derives the value $k_{draw}(p)$, such that *Connect*($k_{draw}(p)$, p) is drawn, as follows. (1) $k_{draw}(2) = 11$. (2) For all $p \geq 3$, $k_{draw}(p) = 3p+3d+8$, where d is a logarithmic function of p. So, the ratio $k_{draw}(p)/p$ is approximate to 3 for sufficiently large p. To our knowledge, our $k_{draw}(p)$ are currently the smallest for all $2 \leq p < 1000$, except for $p = 3$.

1 Introduction

A generalized family of k-in-a-row games, called *Connect*(m, n, k, p, q), were introduced and presented by Wu et al. [9, 10]. Two players, named *Black* and *White*, alternately place p stones on empty *squares*[2] of an $m \times n$ board in each turn, except for that Black plays first and places q stones initially. The player who obtains k consecutive stones of his own first wins. Both players tie when the board is filled up without one winning. For example, *Tic-tac-toe* is *Connect*(3, 3, 3, 1, 1), *Go-Moku* in the free style (a traditional five-in-a-row game) is *Connect*(15, 15, 5, 1, 1), and *Connect6* [10] played on the traditional Go board is *Connect*(19, 19, 6, 2, 1).

In the past, many researchers were engaged in understanding the theoretical values of *Connect*(m, n, k, p, q) games. Allis et al. [1, 2] solved *Go-Moku* with Black winning. Van den Herik et al. [6] and Wu et al. [9, 10] also mentioned several solved games for *k*-in-a-row games.

This paper is interested in drawn *Connect*(m, n, k, p, q) games, where both players have no winning strategy. More specifically, this paper only focuses on *Connect*(∞, ∞, k, p, p) games, denoted by *Connect*(k, p) in this paper. Following strategy-stealing arguments raised by Nash (cf. [3]), Wu et al. [10] showed that White has no winning strategy. In order to prove whether games are drawn, we only need to show that Black has no winning strategy either. Given p, this paper derives the value $k_{draw}(p)$, such that

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

[2] Practically, stones are placed on empty intersections of Renju or Go boards. In this paper, when we say squares, we mean intersections.

Connect($k_{draw}(p)$, p) is drawn. Since a drawn *Connect*(k, p) game also implies a drawn *Connect*($k+1$, p), the value $k_{draw}(p)$ should be as small as possible.

In the past, Zetters [11] derived that *Connect*(8, 1) is drawn. Pluhar derived tight bounds $k_{draw}(p) = p+\Omega(log_2 p)$ for all $p \geq 1000$ (cf. Theorem 1 in [8]). However, the requirement of $p \geq 1000$ is unrealistic in real games. Thus, it becomes important to obtain tight bounds when $p < 1000$. Recently, Hsieh and Tsai [7] derived that $k_{draw}(p) = 4p+7$ for all positive p. So, the ratio $R = k_{draw}(p)/p$ is approximate to 4.

In this paper, Theorem 1 (below) shows that $k_{draw}(2) = 11$, while the result in [7] is 15. Theorem 2 derives a general bound $k_{draw}(p) = 3p+3d+8$ for all $p \geq 3$, where d is a logarithmic function of p, namely $P(d-1) < p \leq P(d)$ and $P(d) = 6\times 2^d - d - 4$. When compared with [7], our $k_{draw}(p)$ are smaller for all $p \geq 5$, and the same for $p = 4$. The ratio $R = k_{draw}(p)/p = 3+(3d+8)/p$ is approximated to 3, which is smaller than 4 in [7]. The proofs of both Theorem 1 and Theorem 2 are given is Section 2 and Section 3, respectively. Some open problems are given in Section 4.

Theorem 1. As described above, *Connect*(11, 2) is drawn. ▌

Theorem 2. Consider all $p \geq 3$. Let $P(d-1) < p \leq P(d)$, where $P(d) = 6\times 2^d - d - 4$. Then, *Connect*($3p+3d+8$, p) are drawn. ▌

2 Proof of Theorem 1

Before proving Theorem 1, we define a new game, called a *ConnectLine game*, as defined in Definition 1.

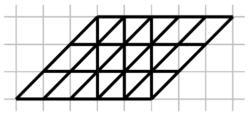


Fig. 1. The game board B_2

Definition 1. On a *game board B* as in *Connect*(k, p), a set of vertically, horizontally and diagonally straight lines are designated, marked as solid lines as illustrated in Fig. 1. Given such a game board B, the game *ConnectLine*(B, p) is defined as follows.

1. The game rules are the same as *Connect*(k, p), except for the following.
2. Black is allowed to place p' stones on B, where $p' \leq p$. In next turn, White is allowed to place p'' stones, where $p'' \leq p'$.
3. Black wins when for some line all the squares of it are occupied by black stones.

The game *ConnectLine*(B, p) is drawn if Black has no winning strategy[3], that is, White has some strategy such that Black cannot win in all cases. ▌

The game boards described in Definition 1 can be viewed as *hypergraphs* [3, 5]. All squares are vertices, while all solid lines are so-called *hyperedges*. The goal of Black is to win by occupying all vertices of some hyperedge.

[3] Based on the strategy-stealing argument, White has no winning strategy.

Let B_2 denote the game board shown in Fig. 1. Lemma 1 (below) shows that *ConnectLine*(B_2, *2*) is drawn. From Lemma 1, Theorem 1 is satisfied for the following reason.

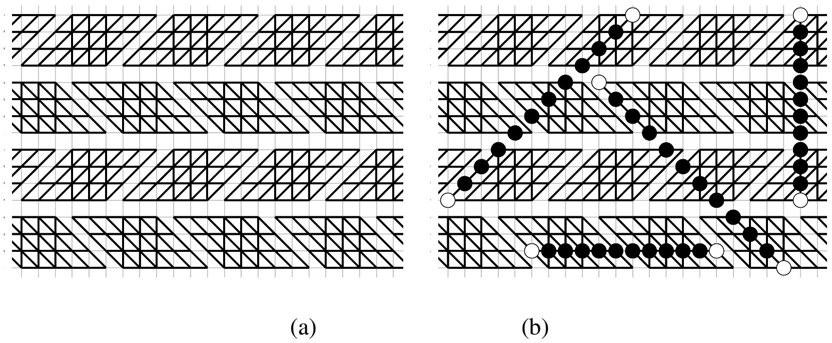
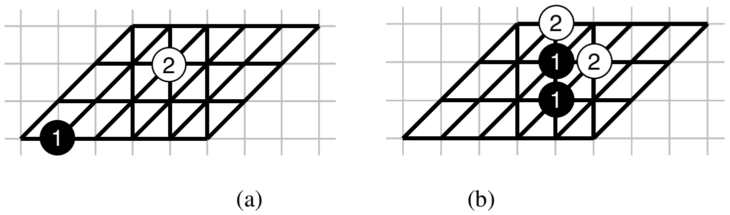

(a) (b)

Fig. 2. (a) Partitioning the infinite board into disjoint B_2. (b) Covering one complete solid line in each segment of 11 consecutive black stones.

First, carefully partition the infinite board into an infinite number of disjoint B_2 (without overlap and vacancy) as shown in Fig. 2 (a). Then, for White, follow the strategy on each B_2 as in Lemma 1, such that none of the solid lines are occupied by all black stones. From Fig. 2 (b), we observe that all segments of consecutive *11* squares vertically, horizontally, and diagonally must cover one complete solid line among these B_2. Since none of these solid lines are occupied by all black stones, none of these segments contains all 11 black stones. Thus, *Connect*(*11*, *2*) is drawn. ▌

Lemma 1. As described above, *ConnectLine*(B_2, *2*) is drawn.



(a) (b)

Fig. 3. (a) White's first defensive moves in Case 1, and (b) in Case 2.1

Proof. A program was written to verify that none of the solid lines in B_2 are occupied by all black stones. ▌

In the remainder of this section, we simply give an intuition for the correctness of a Lemma 1. The first black move is classified into the following cases.

1. Black only places one stone in the board as illustrated in Fig. 3 (a).
2. Black places two stones.

2.1. Both are placed on the middle two squares as those marked “1” in Fig. 3 (b).
2.2. One and only one is placed on either of the two middle squares.
2.3. Neither of the two stones is placed on the two middle squares.

In Case 2.1, White replies two stones as shown in Fig. 3 (b); and in all the other cases, place one stone on one of the middle two squares. Here, we only illustrate Case 1 in Fig. 3 (a) and Case 2.1 in Fig. 3 (b).

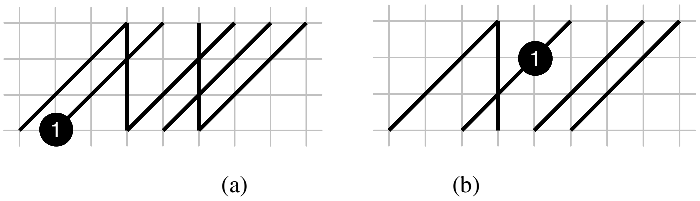


Fig. 4. The active vertical and diagonal lines after White’s first move. (a) Case 1. (b) Case 2.1.

After the first White move is made, Fig. 4 shows the boards with active vertical and diagonal lines only. An *active line* is a line without any white stones yet. A game board is called a *tree-based game board* (or simply a tree in a hypergraph [3, 5]), if all the solid lines form no cycles in the board as illustrated in both cases in Fig. 4. Lemma 2 (below) shows that a game is drawn if its game board is tree-based with line lengths no more than four and with at most one black stone. Thus, from Lemma 2, the two games in Fig. 4 are drawn.

Lemma 2. Assume that there exists at most one black stone on a tree-based game board B_T and that all the line lengths in B_T are no more than four. Then, *Connect-Line*(B_T, *2*) is drawn.

Proof. Assume that there exists one black stone on some square *s*. Let Black’s next move place one stone on another square *s'*. Since the game board is tree-based, we find at most one path (a sequence of lines) from *s* to *s’* and then place one stone on one of these lines in the path, if any. (Note that if both *s* and *s'* are on the same line, White simply places on that line.) Thus, B_T is broken into two tree-based game boards or more, each of which contains at most one black stone. If Black’s next move places two stones, simply use two stones to break the game board into three or more as above. Thus, this lemma holds by induction. ▌

However, for Lemma 1, the proof still needs to exclude the case that some horizontal line is occupied by all black stones. The proof for this becomes tedious. In practice, we wrote a program to prove it by exhaustedly searching all cases. The details are omitted in this paper.

3 Proof of Theorem 2

In this proof, similar to that of Theorem 1, the infinite board is partitioned into an infinite number of disjoint game boards $B_Z(L)$ (without overlap and vacancy) as shown in Fig. 5 (below). The game board $B_Z(L)$ is shown in Fig. 6 (a) (below), where all the lengths of solid lines are *L* and the game board extends infinitely to both sides.

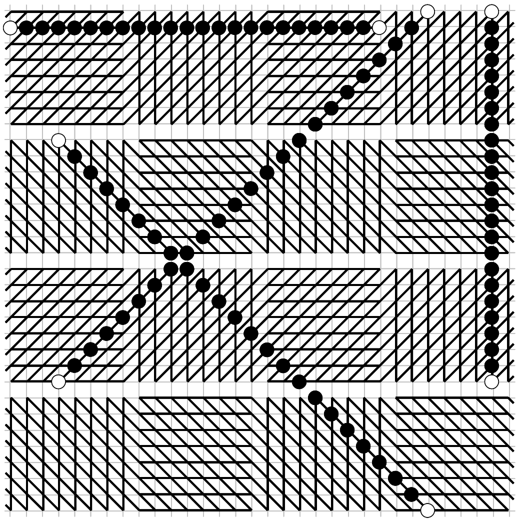
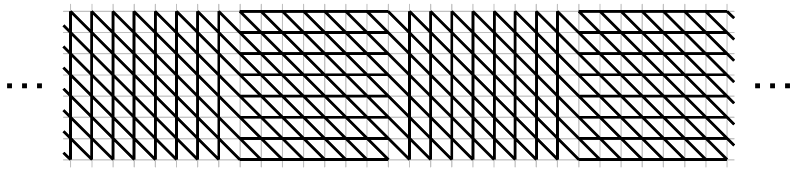
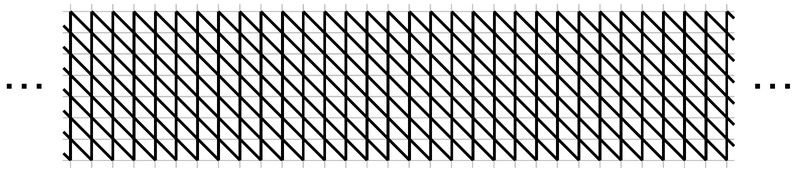


Fig. 5. Partitioning the infinite board into disjoint $B_Z(L)$



(a) $B_Z(L)$



(b) $B_N(L)$

Fig. 6. Two game boards, $B_Z(L)$ and $B_N(L)$

Similarly, we observe from Fig. 5 that all segments of $3L–1$ consecutive squares vertically, horizontally, and diagonally must cover one solid line among these $B_Z(L)$. Assume that the game *ConnectLine*($B_Z(L)$, *p*), also denoted by *ConnectBZ*(*L*, *p*) for simplicity, is drawn, that is, White has some strategy such that none of the solid lines in $B_Z(L)$ are occupied by all black stones. Thus, by following this strategy on each $B_Z(L)$, White prevents Black from occupying any segment of $3L–1$ consecutive squares completely. Thus, *Connect*($3L–1$, *p*) is drawn and Corollary 1 is satisfied.

Corollary 1. As described above, if *ConnectBZ*(*L*, *p*) is drawn, then *Connect*($3L–1$, *p*) is drawn. ▌

The rest of the proof is outlined as follows. In Subsection 3.1, we show that the game board $B_Z(L)$ is *isomorphic* to $B_N(L)$ as shown in Fig. 6 (b), in the sense of *hypergraph isomorphism* [3, 5]. Most importantly, from this, Subsection 3.1 shows that Corollary 2 is satisfied. Similarly, let *ConnectBN*(L, p) denote the game *ConnectLine*($B_N(L)$, p). In the rest of the subsections, we prove that Lemma 4 (below) holds. Thus, Theorem 2 is satisfied from Corollary 2 and Lemma 4. For simplicity of discussion, Subsection 3.2 first proves Lemma 3, simplified from Lemma 4. Subsection 3.3 then completes the proof of Lemma 4. ▌

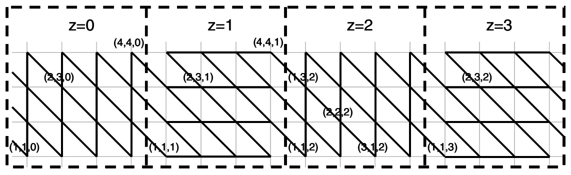
Corollary 2. As described above, if *ConnectBN*(L, p) is drawn, then *Connect*($3L–1$, p) is drawn. ▌

Lemma 3. For all $p = 5\times2^d–d–4$, where $d \geq 0$, *ConnectBN*($p+d+3$, p) are drawn. ▌

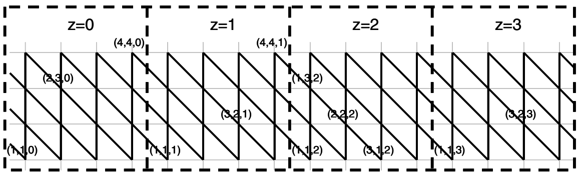
Lemma 4. Consider all $p \geq 1$. Let $P(d–1) < p \leq P(d)$, where $P(d) = 6\times2^d–d–4$. Then, *ConnectBN*($p+d+3$, p) are drawn. ▌

3.1 Isomorphism

Both game boards $B_Z(L)$ and $B_N(L)$ are hypergraph isomorphic [3, 5] by the following mapping. Let every L neighboring vertical or horizontal solid lines be grouped into one zone in both $B_Z(L)$ and $B_N(L)$ as shown in Fig. 7 (a) and (b) respectively. In both game boards, each square is set to a coordinate (x, y, z), where the square is in the x'th column (from left) and in the y'th row (from bottom) in zone z. Let each square at (x, y, z) on $B_Z(L)$ be mapped into the one at (x, y, z) on $B_N(L)$ when z is even, and at (y, x, z) on $B_N(L)$ when z is odd. Let all solid lines (or hyperedges) on $B_Z(L)$ be mapped into those on $B_N(L)$ accordingly, except for that the i'th horizontal line (from bottom) on $B_Z(L)$ is mapped to the i'th vertical line (from left).



(a) $B_Z(4)$



(b) $B_N(4)$

Fig. 7. Coordinate flipping between $B_Z(4)$ and $B_N(4)$

Lemma 5. Consider both *ConnectBZ*(L, p) and *ConnectBN*(L, p) games over all L and p. Then, *ConnectBZ*(L, p) is drawn if and only if *ConnectBN*(L, p) is drawn.

Proof. According to the above mapping from $B_Z(L)$ to $B_N(L)$, placing one stone on (x, y, z) in $B_Z(L)$ is equivalent to placing one stone on (y, x, z) in $B_N(L)$, and *vice versa*. Since both $B_Z(L)$ and $B_N(L)$ are hypergraph isomorphic for the mapping, some solid line (hyperedge) of $B_Z(L)$ is occupied by all black stones, if and only if the mapped solid line of $B_N(L)$ is. Therefore, *ConnectBZ*(L, p) is drawn if and only if *ConnectBN*(L, p) is drawn. ▌

Corollary 2 is satisfied directly from Corollary 1 and Lemma 5.

3.2 Proof of Lemma 3

Lemma 3 is proved by induction. Lemma 6 (below) shows the initial case that *ConnectBN*(4, 1) is drawn. Lemma 7 shows that if *ConnectBN*(L, p) is drawn, then *ConnectBN*($2L+1$, $L+p$) is drawn too. From the two lemmas, Lemma 3 holds. Subsection 3.2.1 proves Lemma 6, while Subsection 3.2.2 proves Lemma 7. ▌

Lemma 6. *ConnectBN*(4, 1) is drawn. ▌

Lemma 7. Assume that *ConnectBN*(L, p) is drawn. Then, *ConnectBN*($2L+1$, $L+p$) is drawn too. ▌

3.2.1 Drawn ConnectBN(4, 1)

Let us shorten the solid lines of $B_N(4)$ into $B_{N-}(4)$ as shown in Fig. 8. Since $B_{N-}(4)$ is a tree-based game board and none of the black stones exists initially, $B_{N-}(4)$ is drawn from Lemma 2. Obviously, this implies that $B_N(4)$ with extra longer lines is drawn too. ▌

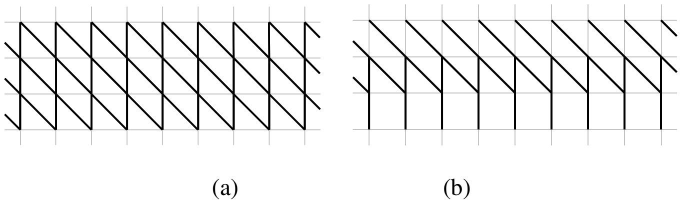


Fig. 8. (a) $B_N(4)$. (b) $B_{N-}(4)$, the same as $B_N(4)$ except for that all the solid lines are shorten.

3.2.2 Induction

In order to prove Lemma 7, we need to consider game boards with extra *exclusive squares* as defined in Definition 2.

Definition 2. In a game board B as described in Definition 1, some of the squares each of which belongs to one distinct line only are designated as *exclusive squares*, as illustrated with solid bullets in Fig. 9 (a) and (b) (below). The game *ConnectLX*(B, b) is the same as *ConnectLine*(B, ∞), except for the following additional rules.

1. Black is excluded to place stones on these exclusive squares.
2. Black wins if some active line contains more than b black stones in Black's turn[4].

The game is drawn if White has a strategy such that Black does not win in all cases. ▌

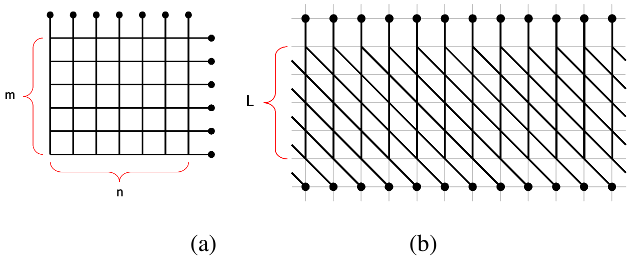


Fig. 9. Two game boards with exclusive squares (solid bullets). (a) $B_{recX}(m, n)$. (b) $B_{NX}(L)$.

In order to prove this theorem, we need to consider the following two game boards. One game board, denoted by $B_{recX}(m, n)$, consists of m horizontal lines and n vertical lines each with one extra exclusive square, as shown in Fig. 9 (a). The other game board, denoted by $B_{NX}(L)$, simply extends each line of $B_N(L)$ by one exclusive square, as shown in Fig. 9 (b). For simplicity of discussion, let *ConnectBNX*(L, b) denote the game *ConnectLX*$(B_{NX}(L), b)$. Both Lemma 8 and Lemma 9 (below) show useful properties related to the two boards, respectively. These properties result in an important lemma, Lemma 10.

From Lemma 9, since *ConnectBN*(L, p) is drawn (from the assumption of Lemma 7), *ConnectBNX*$(L, L–p–1)$ is drawn. Thus, we obtain that *ConnectBN*$(2L+1, 2L–(L–p–1)–1)$ is drawn from Lemma 10, that is, *ConnectBN*$(2L+1, L+p)$ is drawn. Thus, Lemma 7 holds. ▌

Lemma 8. As described above, *ConnectLX*$(B_{recX}(m, n), 1)$ is drawn over all m and n.

Proof. It suffices to prove that White has a strategy such that at most one of the active lines contains black stones and the active line, if it exists, contains at most one black stone in Black's turn. Let variable $R(i)$ and $C(j)$ be the numbers of black stones in the i'th horizontal line and the j'th vertical line respectively, if the lines are still active, and otherwise be 0 (if not active). Let variable N be the summation over all $R(i)$ and $C(i)$. Then, it suffices to prove that White has a strategy such that $N \leq 1$ in all Black's turns.

Assume by induction that $N \leq 1$ in some Black's turn. Assume that Black places only one stone for each move M. Obviously, the move M increases N by at most two. That is, $N \leq 3$. White follows a strategy S to make moves such that $N \leq 1$ as follows.

1. Let Black place one stone on square s at row r and column c.
2. In the case of $N \leq 2$, simply block one active line with some black stone by placing one white stone on the exclusive square in that line. Thus, in this case, N is at most one.

[4] In a game, when we say in Black's turn (White's turn), we mean the moment after White (Black) makes a move and before Black (White) makes the next move.

3. In the case of $N = 3$, if some active line contains two black stones, simply block the active line by placing one white stone on the exclusive square in that line.
4. In the rest case that $N = 3$ and that none of active lines contains two black stones, assume some $R(r') = 1$ where $r' \neq r$ without loss of generality. Thus, the square s' at row r' and column c (both lines are active) must be empty (otherwise, two black stones in the column as at Step 3). Therefore, simply place one white stone on s'. Since the stone blocks the two active lines in row r' and column c, N is back to one.

However, if Black makes a move with several black stones at a time, say t black stones, we separate the move into t sub-moves each with one stone only. Then, White pretends that Black made sub-moves one by one, and therefore follows the above strategy S to place stones, except for the following case. If White is to place one stone on an empty square s' at Step 4 as above, but one of the subsequent sub-moves M' places one black stone on it, the strategy needs to be changed as follows.

5. Place two white stones respectively on the exclusive squares of the two active lines in row r' and column c containing s'. Thus, N is back to one too. Besides, when Black play M', White does not need to place any stones. Thus, the two white stones together are viewed as a reply to the two black stones at sub-moves M and M'.

From the above strategy, $N \leq 1$ is maintained in Black's turn. Thus, this lemma holds. ▌

Lemma 9. As described above, assume that the game *ConnectBN*(L, p) is drawn. Then, *ConnectBNX*(L, $L–p–1$) is drawn.

Proof. Since *ConnectBN*(L, p) is drawn, White has some strategy S such that all active lines have at most $L–p–1$ black stones in Black's turn. Otherwise, for some active line with at least $L–p$ black stones, Black wins by simply placing p stones on this line.

In the game *ConnectBNX*(L, $L–p–1$), if Black still places at most p black stones for all moves, then White simply follows strategy S (without placing stones on exclusive squares) such that all active lines in $B_{NX}(L)$ contains at most $L–p–1$ black stones in all Black's turns. However, if Black makes a move with more than p black stones, we separate the move into several sub-moves, each with at most p black stones. Then, White pretends that Black made sub-moves one by one, and for each sub-move M simply follows S to play with the following exceptional case. White follows S to place one white stone on an empty square s, but some of the subsequent sub-moves M' will place one black stone, the strategy is changed as follows.

1. Place two white stones respectively on the exclusive squares of the two lines containing s, instead. The reason is similar to that in Step 5 in Lemma 8. Since all the lines containing s are no longer active, the stone at s can be ignored in M'. Let the stone at s be added into M and removed from M'. Thus, White's reply to M keeps Black from containing more than $L–p–1$ black stones. Although White's reply uses one more stone, sub-move M has one more stone too.

Thus, all active lines in the game *ConnectBNX*(L, $L–p–1$) still have no more black stones than those in the game *ConnectBN*(L, p). That is, all active lines in the game

ConnectBNX(*L*, *L–p–1*) have at most *L–p–1* black stones. That is, *ConnectBNX*(*L*, *L–p–1*) is drawn. ▌

Lemma 10. Assume that *ConnectBNX*(*L*, *b*) is drawn, where $0 < b < L$. Then, *ConnectBN*(*2L+1*, *2L–b–1*) is drawn too.

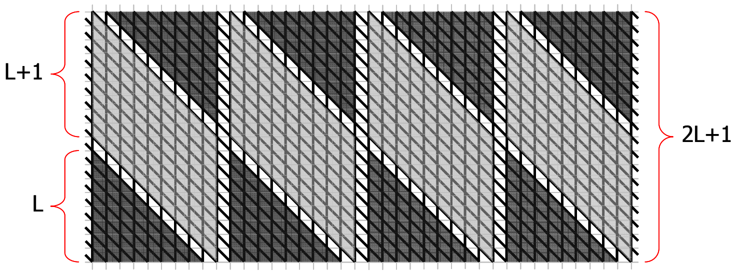


Fig. 10. Dividing $B_N(2L+1)$ into dark and light gray zones

Proof. It suffices to prove that White has some strategy such that all active lines in $B_N(2L+1)$ contain at most $b+1$ black stones in all Black's turns. Now, divide $B_N(2L+1)$ into two zones, dark and light gray zones, as shown in Fig. 10. We want to prove the property: all active lines in $B_N(2L+1)$ contain at most *one* black stone in the light gray zone and at most *b* in the dark gray zone in all Black's turns. By induction, we assume that this property is satisfied in Black's last turn. Since Black places at most $2L–b–1$ black stones in the next move, all active lines contain at most $(2L–b–1)+(b+1) = 2L$ ($< 2L+1$) black stones. Thus, all active lines still have empty squares in White's turn, which will be used as exclusive squares, whenever needed below.

In the light gray zone, each parallelogram, a $(L+1)\times(L+1)$ matrix of squares, is transformed into $B_{recX}(L+1, L+1)$ by adding an extra exclusive square into each active line. Let White follow the strategy given in Lemma 8 to defend in the parallelogram. (Note that White places on an empty square in the corresponding active line in $B_N(2L+1)$ whenever required to place on the exclusive squares described in Lemma 8.) Thus, from Lemma 8, at most one of the active lines contains black stones and at most one black stone in the parallelogram in Black's next turn. In brief, each active line in the parallelogram ($B_{recX}(L+1, L+1)$) contains at most one black stone.

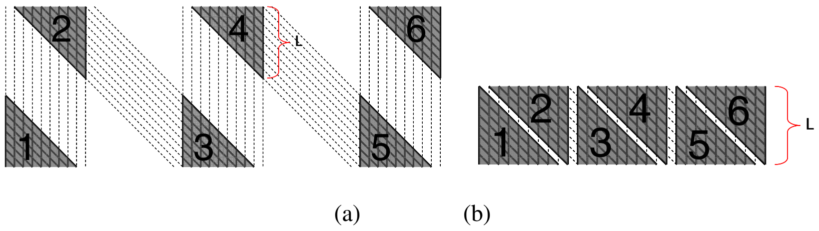


Fig. 11. (a) Half of the dark gray zone. (b) Squeezing the zone in (a) into a $B_N(L)$.

In the dark gray zone, we consider each half of them as shown in Fig. 11 (a), and then squeeze them into a $B_N(L)$ game board in Fig. 11 (b). The squeezed $B_N(L)$ is transformed into $B_{NX}(L)$ by adding an extra exclusive square into each active line. From the assumption of this lemma, White has a strategy in game *ConnectBNX*(L, b) such that all active lines in $B_{NX}(L)$ contain at most b black stones in Black's turn. Similarly, White can place on any empty square in the corresponding active line in $B_N(2L+1)$ whenever placing on the exclusive squares in the strategy. It is concluded that each active line in the dark gray zone contains at most b black stones. ∎

3.3 Proof of Lemma 4

In order to solve all p, we first investigate some small p and, second, generalize both Lemma 7 and Lemma 10 into Lemma 11 and Lemma 12 (below) as follows. First, both *ConnectBNX*(2, 1) and *ConnectBNX*(3, 2) are drawn from Lemma 13, and *ConnectBNX*(4, 2) is drawn from both Lemma 6 and Lemma 9. From these drawn *ConnectBNX* games and Lemma 12, we obtain that *ConnectBN*(5, 2), *ConnectBN*(7, 3) and *ConnectBN*(8, 4) are drawn. Then, from the above drawn *ConnectBN* games, we derive that Lemma 4 holds for all $p > 4$ by applying Lemma 11 recursively. The details are omitted. ∎

Lemma 11. Assume that both *ConnectBN*(L_1, p_1) and *ConnectBN*(L_2, p_2) are drawn. Let $p' = \min(L_2+p_1, L_1+p_2)$. Then, *ConnectBN*($L_1+L_2+1$, p') is drawn too.

Proof. Since both *ConnectBN*(L_1, p_1) and *ConnectBN*(L_2, p_2) are drawn, both *ConnectBNX*(L_1, L_1-p_1-1) and *ConnectBNX*(L_2, L_2-p_2-1) are drawn from Lemma 9. From Lemma 12 (below), *ConnectBN*(L_1+L_2+1, p') is drawn too, since $L_1+L_2-\max(L_1-p_1-1, L_2-p_2-1)-1 = \min(L_2+p_1, L_1+p_2) = p'$. ∎

Lemma 12. Assume that both *ConnectBNX*(L_1, b_1) and *ConnectBNX*(L_2, b_2) are drawn, where $0 < b_1 < L_1$ and $0 < b_2 < L_2$. Let b' be $\max(b_1, b_2)$. Then, *ConnectBN*(L_1+L_2+1, $L_1+L_2-b'-1$) is drawn too.

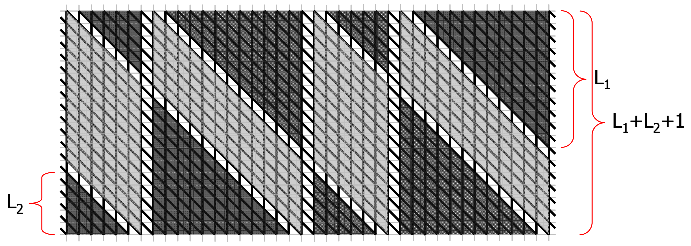


Fig. 12. Dividing $B_N(L_1+L_2+1)$ into dark and light gray zones

Proof. This proof is similar to that in Lemma 10. Divide $B_N(L_1+L_2+1)$ into dark and light gray zones, as shown in Fig. 12. First, consider the active lines covering the dark gray zones with larger triangles in $B_N(L_1+L_2+1)$. For the same reason described in Lemma 10, each of the active lines contains at most one black stone in the light and at most b_1 in the dark in all Black's turns. Similarly, for the active lines covering the dark with smaller triangles in $B_N(L_1+L_2+1)$, each of the active lines contains at most one in the light and at most b_2 in the dark in all Black's turns. Thus, each active line contains at most $1+\max(b_1, b_2) = 1+b'$ black stones in all Black's turns. Since Black places at most $L_1+L_2-b'-1$ black stones, Black does not connect to L_1+L_2+1. Thus, *ConnectBN*(L_1+L_2+1, $L_1+L_2-b'-1$) is drawn. ▌

Lemma 13. Both *ConnectBNX*(*2*, *1*) and *ConnectBNX*(*3*, *2*) are drawn.

Proof. Omitted in this paper. ▌

4 Future Work

This paper presents tighter bounds $k_{draw}(p)$ for $5 \le p < 1000$ and $p = 2$. More problems are still open and are as follows.

- Derive lower $k_{draw}(p)$ for $p < 1000$, especially for small p, e.g., $1 \le p \le 10$. These problems are more realistic in real games.
- Derive general tight bounds that are smaller than those in this paper and those in [8] simultaneously.

Acknowledgments

The authors would like to thank the National Science Council of the Republic of China (Taiwan) for financial support of this research under contract numbers NSC 95-2221-E-009-122-MY2 and NSC 97-2221-E-009-126-MY3. The authors would also like to thank Po-Ting Chen for some helps about the program in Theorem 1 and anonymous referees for valuable comments.

References

1. Allis, L.V.: Searching for solutions in games and artificial intelligence. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands (1994)
2. Allis, L.V., Van den Herik, H.J., Huntjens, M.P.H.: Go-Moku solved by new search Techniques. Computational Intelligence 12, 7–23 (1996)
3. Berge, C.: Graphs and Hypergraphs. North Holland, Amsterdam (1973)
4. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for your Mathematical Plays, 2nd edn., vol. 3. A K Peters. Ltd., Canada (2003)
5. Diestel, R.: Graph Theory, 2nd edn. Springer, New York (2000)
6. Van den Herik, H.J., Uiterwijk, J.W.H.M., Rijswijck, J.V.: Games solved: now and in the future. Artificial Intelligence 134, 277–311 (2002)
7. Hsieh, M.-Y., Tsai, S.-C.: On the fairness and complexity of generalized k-in-a-row games. Theoretical Computer Science 385, 88–100 (2007)
8. Pluhar, A.: The accelerated k-in-a-row game. Theoretical Computer Science 271, 865–875 (2002)
9. Wu, I.-C., Huang, D.-Y.: A New Family of k-in-a-row Games. In: The 11th Advances in Computer Games (ACG11) Conference, Taipei, Taiwan (2005)
10. Wu, I.-C., Huang, D.-Y., Chang, H.-C.: Connect6. ICGA Journal 28(4), 234–242 (2006)
11. Zetters, T.G.L.: 8(or more) in a row. American Mathematical Monthly 87, 575–576 (1980)

Optimal Analyses for $3 \times n$ AB Games in the Worst Case

Li-Te Huang and Shun-Shii Lin

Department of Computer Science and Information Engineering,
National Taiwan Normal University, No. 88, Sec. 4,
Ting-Chow Rd., Taipei, Taiwan, ROC
linss@csie.ntnu.edu.tw

Abstract. The past decades have witnessed a growing interest in research on deductive games such as Mastermind and AB game. Because of the complicated behavior of deductive games, tree-search approaches are often adopted to find their optimal strategies. In this paper, a generalized version of deductive games, called $3 \times n$ AB games, is introduced. However, traditional tree-search approaches are not appropriate for solving this problem since it can only solve instances with smaller n. For larger values of n, a systematic approach is necessary. Therefore, intensive analyses of playing $3 \times n$ AB games in the worst case optimally are conducted and a sophisticated method, called *structural reduction*, which aims at explaining the worst situation in this game is developed in the study. Furthermore, a worthwhile formula for calculating the optimal numbers of guesses required for arbitrary values of n is derived and proven to be final.

1 Introduction

With the rapid increase in the need of encryption, it becomes urgent to develop an efficient mechanism of cryptanalysis. A kind of cryptanalysis, *differential cryptanalysis*, however, bears resemblance to deductive games in accordance with the analysis by Merelo-Guervos *et al.* [1]. In other words, deductive games can be regraded as abstract models of cryptanalysis problems and any results of deductive games may be applied to cryptography or related combinatorial optimization problems.

There are two players invloved in a deductive game. They are called the *codemaker* and the *codebreaker*, respectively. In the beginning of the game, the codemaker thinks of a secret code in mind and afterwards, the codebreaker tries to identify this secret code by guessing continuously. As long as the codebreaker makes a guess, the codemaker will give him[1] a response to describe the similarity between this guess and the secret code. The mission of the codebreaker is to obtain the code and minimize the number of guesses required at the same time.

More precisely, an $m \times n$ deductive game means that each possible secret code in the game is composed of m digits while every digit has n possibilities

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

(symbols). Without loss of generality, the set of these n symbols is defined as $S = \{0, 1, 2, ..., n-1\}$. Assumee that the codemaker has a secret code $c = c_1c_2...c_m$ in mind and the codebreaker makes a guess $g = g_1g_2...g_m$, where $s_i, g_j \in S, \forall i, j$. Then, the codemaker will give a response $[A, B]$, where A and B are defined as follows.

- $A = |\{i : c_i = g_i\}|, \forall i = 1, ..., m$. Thus, A means the number of symbols which appear in both c and g and meanwhile, every symbol occupies the same position in both c and g.
- $B = \sum_{j=0}^{n} \min(p_j, q_j) - A$, where $p_j = |\{i : c_i = j\}|$ and $q_j = |\{i : g_i = j\}|$. In other words, B represents the number of symbols which occur in both c and g but the positions of these symbols in c and g do not match.

Besides the above definitions, there is one additional characteristic to distinguish two families of deductive games. One of the two is Mastermind, in which repeated symbols are allowed in a secret code. The other is AB game, in which all symbols within a code are distinct. Note that the numbers of all possible responses given by the codemaker and all possible guesses the codebreaker can make are calculated as follows.

- For an $m \times n$ deductive game, the codemaker may give one of these responses which are $[m, 0], [m-1, 0], [m-2, 2], [m-2, 1], [m-2, 0], ..., [m-i, i], ..., [m-i, 0], ..., [0, m], ..., [0, 0]$. So, the total number of responses is $1 + 1 + 3 + 4 + 5 + ... + (m+1) = m(m+3)/2$.
- Obviously, there are n^m secret codes in Mastermind and $n!/(n-m)!$ codes in AB game.

In this study, $3 \times n$ AB games are investigated and analyzed, i.e., the case of $m = 3$ is discussed here. Hence, the number of all possible responses is 9 and these responses are [3, 0], [2, 0], [1, 2], [1, 1], [1, 0], [0, 3], [0, 2], [0, 1], and [0, 0] respectively. The number of all possible secret codes equals to $n(n-1)(n-2)$ as well.

For example, assume that the codemaker choose $c = 215$ as a secrete code and meanwhile, the codebreaker makes a guess $g = 012$. Then, the codemaker will offer a response $[1, 1]$.

This paper consists of six major parts. In Section 2, previous surveys are conducted. Section 3 provides some terminologies and notations. The optimal guess for the codebreaker in each turn is analyzed in Section 4. The worst situation caused by the codemaker is discussed in Section 5. Section 6 derives a theorem, which can calculate the optimal number of guesses for $3 \times n$ AB games in the worst case and some conclusions are also given.

2 Preliminaries

Two well-known deductive games are Mastermind and AB game, of which the dimensions are 4×6 and 4×10, respectively. The former is popular in America while the later is widespread in England and Asia. AB game is called "Bulls and Cows" in some places as well.

There have been much research on Mastermind and AB game over past several decades since Knuth [2] first investigated them. Knuth also proposed a worst-case optimal strategy of Mastermind, where the maximum number of guesses is 5. Meanwhile, its expected-case number of guesses is 4.478. Later, many studies for finding better strategies of Mastermind in the expected case were conducted. For example, Irving [3], Neuwirth [4], and Norvig [5] improved the expected-case strategies, in which the required guesses are 4.369, 4.364, and 4.47 in average, respectively. Koyama and Lai [6] demonstrated an optimal strategy in the expected case for Mastermind eventually while the expected number of guesses is about 4.34. Rosu [7] subsequently proposed an alternative algorithm to obtain its optimal strategy as well. Afterwards, a new heuristics for Mastermind was suggested by Barteld [8] and outperformed the conventional heuristics. Recently, an advanced framework to seek the optimal strategy for Mastermind was suggested by Huang *et al.* [9] as well, and this algorithm is innately superior to traditional ones since branch-and-bound pruning is adopted. There were also other approaches that emphasized the efficiency of determining good strategies such as Shapiro [10] and Rosu [7]. However, the qualities of solutions may be often worse than those of sophisticated methods because the simple approaches may not take sufficient time to consider all possible strategies carefully. In research on 4×10 AB game, Chen *et al.* [11] first obtained an optimal strategy in the worst case and showed that the maximum number of guesses is 7.

For the deductive games with higher dimensions, meta-heuristic algorithms are usually developed to solve them. For instance, Kalisker and Camens [12], Singley [13], Chen *et al.* [14], and Berghman *et al.* [15] proposed several meta-heuristic approaches to deal with Mastermind with different dimensions. One crucial research worthy of mention is Chen *et al.* [14]. This might be a promising result as it is the first approximate approach to achieve a near-optimal result for 4×6 Mastermind in the expected case. Although these methods often operate efficiently and effectively, they are not able to guarantee to attain optimal strategies.

For those deductive games with smaller dimensions, tree-search approaches or heuristics are often adopted to find their optimal strategies. However, these common methods are not appropriate for solving deductive games with higher dimensions. Hence, a systematic theoretical analysis, called *graph-partition approach*, was used by Chen *et al.* to investigate $2 \times n$ AB games [16] and $2 \times n$ Mastermind [17]. Optimal results of the two games in the worst and expected case were obtained eventually. Goddard [18] offered some basic discussions of $m \times n$ Mastermind and also attained the optimal numbers of guesses for $2 \times n$ Mastermind in both the worst and expected case, which were basically the same as those in Chen *et al.* [17].

3 Terminologies

Before $3 \times n$ AB games is discussed formally, some terminologies and notations have to be explained first in order to describe the analyses precisely. Thus, some terms are defined as follows.

Definition 1. *A secret code is* ***eligible*** *if it is compatible with all guesses and the corresponding responses given so far.*

Definition 2. *A set, which contains some eligible codes, is referred to as a* ***state****.*

Definition 3. *The state with only one eligible code, which has also been guessed by the codebreaker now, is defined as a* ***final state****. That is to say that the secret code has been identified and the game is over.*

Definition 4. *Let C_1 and C_2 denote two states. We say that C_1 is* ***harder*** *than C_2 if identifying a secret code in C_1 requires more guesses than that in C_2. In other words, the* ***difficulty*** *of a state means how many guesses the codebreaker requires to identify a secret code.*

Definition 5. *A strategy of responses taken by the codemaker is called a* ***devil's strategy*** *or an* ***adversary response*** *if this strategy maximizes the number of guesses required by the codebreaker.*

Definition 6. *Assume that there are two states, which are C_1 and C_2 respectively. If there exists a one-to-one function r such that each secret code in C_1 maps another one in C_2 and preserves the structure of C_1, then we say that C_2* ***dominates*** *C_1. Furthermore, r is called a* ***structural reduction****. In symbols, we write $C_1 \leq C_2$.*

Now, 3×5 AB game is taken into account as an illustrative example. Assumee that the set of five symbols in this game is $S = \{0, 1, 2, 3, 4\}$. If the codebreaker makes a guess, 012, and the codemaker responses [2, 0] in the first turn, the eligible codes are therefore 013, 014, 032, 042, 312, and 412 after the first turn. The set $C_{[2,0]} = \{013, 014, 032, 042, 312, 412\}$ forms a state. From the result of the later experiment, which conducts an exhaustive search to 3×5 AB game, the number of guesses required is maximum if the codemaker implements a devil's strategy to provide the response, [0, 2], at the first response. In contrast, $C_{[2,0]}$ and the state, $C_{[1,0]} = \{043, 034, 432, 342, 314, 413\}$, which is produced when the codemaker responses [1, 0] at the first response, are then considered. Notice that the elements in $C_{[2,0]}$ are of the forms, $01b$, $0b2$, or $b12$, where $b \in B = \{3, 4\}$. Thus, we define a structural reduction of r as

$$r : \begin{cases} 01b \mapsto 0zb \\ 0b2 \mapsto zb2, \text{ where } b \in B \text{ and } z \in B - \{b\} . \\ b12 \mapsto b1z \end{cases}$$

Figure 1 exhibits the mapping of each code in $C_{[2,0]}$ in detail. Note that the mapped codes in $C_{[1,0]}$ preserve the structures of those in $C_{[2,0]}$. This implies that finding a secret code in $C_{[1,0]}$ is *as hard as* or *harder* than that in $C_{[2,0]}$. Intuitively, this is also obvious since there is one more identified symbol in $C_{[2,0]}$ than in $C_{[1,0]}$. Hence, we say that $C_{[1,0]}$ dominates $C_{[2,0]}$. Furthermore, the structural reduction has the property of the transitive relation obviously. That is to say that $C_1 \leq C_3$ if $C_1 \leq C_2$ and $C_2 \leq C_3$.

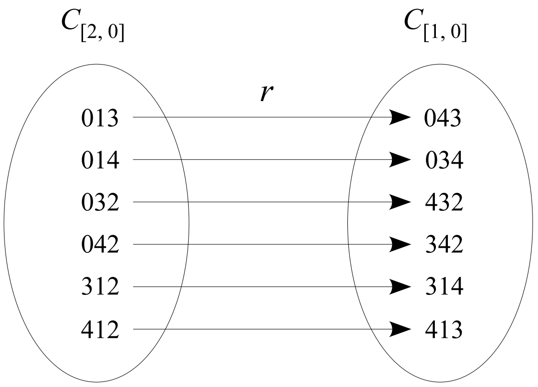


Fig. 1. Mapping from codes in $C_{[2,0]}$ to those in $C_{[1,0]}$ for 3×5 AB game

4 Analyses of the Optimal Guesses for the Codebreaker

In this section, the analyses of the optimal guess in each turn for the codebreaker are provided. First, a special kind of states C^* will be analyzed to determine the best guess for the codebreaker when he encounters this kind of states. Then, the discussion in the next section will reveal that the special states that are discussed here just match the attribution of states resulting from the devil's strategy for the codemaker. Consequently, our conclusions are attained finally.

Before the formal discussion, a critical concept should be clarified first. Intuitively, the more secret codes a state has, the harder the codebreaker identifies a secret code in it. However, the rule is not absolutely correct especially when the size of one state is very close to that of the other. Hence, the structural reduction is adopted to determine the difficulties of two states instead of simply comparing their sizes in the following discussion.

Assume that $S = \{0, 1, 2, ..., n-1\}$ represents the set of symbols appearing in $3 \times n$ AB games. The set, $B = \{b_0, b_1, ..., b_{h-1}\}$, is a subset of S, where $b_i \in S$ and $|B| = h, 3 \leq h \leq n-3$. Moreover, another set, A, is defined as $A = S - B = \{a_0, a_1, ..., a_{n-h-1}\}$, of which the cardinality is $(n-h)$.

Assume that there is a special state, called C^*, which consists of the secret codes that are all possible permutations of h symbols in B. In other words, the special state has $h\,(h-1)\,(h-2)$ secret codes in it. This state may be regarded as a subproblem of a $3 \times n$ AB game, i.e., a $3 \times h$ AB game. Notice that the symbols in A do not appear in the codes of the special state because of the definition of C^*. We can intuitively treat the symbols in A as those eliminated from previous responses made by the codemaker.

Now, imagine a scenario where C^* is encountered by the codebreaker during the process of playing a $3 \times n$ AB game. Since any symbols in S may be used in a guess made by the codebreaker for a $3 \times n$ AB game, all possible guesses for the codebreaker can be classified into four types according to the numbers of symbols that belong to A and B. Thus, the four types of guesses for the codebreaker are listed and discussed as follows. Here we assume that $a_i, a_j, a_k \in A$ and $b_i, b_j, b_k \in B$.

1. $a_i a_j a_k$
   All symbols of this type of guesses belong to A. If the codebreaker makes this kind of guesses, all eligible codes are then classified into the substate, $C_{[0,0]}$, trivially. So, the guesses of Type 1 are redundant and non-optimal results will be obtained if the codebreaker chooses this kind of guesses.
2. $b_k a_i a_j, a_i b_k a_j,$ and $a_i a_j b_k$
   The guesses of Type 2 contain two symbols in A and one symbol in B. This type of guesses can be further divided into three kinds of guesses such as $b_k a_i a_j, a_i b_k a_j,$ and $a_i a_j b_k$ in accordance with their positions of symbols. Without loss of generality, $g = b_k a_i a_j$ is taken to conduct the following analyses. The discussions of the other two can be undertaken in a similar way. Three nonempty substates, which are $C_{[1,0]}, C_{[0,1]},$ and $C_{[0,0]}$, are produced as the codebreaker makes the guess, g. Note that their cardinality are $(h-1)(h-2), 2(h-1)(h-2),$ and $(h-1)(h-2)(h-3)$, respectively. Now, we can show that $C_{[0,1]} \leq C_{[0,0]}$ and $C_{[1,0]} \leq C_{[0,0]}$ if $h \geq 5$.

   Lemma 1. *If the codebreaker encounters the state, C^*, and then makes the guess, $g = b_k a_i a_j, a_i b_k a_j$, or $a_i a_j b_k$, where $a_i, a_j \in A$ and $b_k \in B$, then $C_{[0,0]}$ dominates $C_{[0,1]}$ and $C_{[1,0]}$ if $h \geq 5$.*

   Proof. In order to prove that $C_{[0,1]} \leq C_{[0,0]}$, a structural reduction, r_1, is defined as

$$r_1 : \begin{cases} b_p b_k b_q \mapsto b_p z_1 b_q, \text{ where } b_p, b_q \in B' = B - \{b_k\} \\ b_p b_q b_k \mapsto b_p b_q z_2 \quad \text{and } z_1, z_2 \in B' - \{b_p, b_q\} . \end{cases}$$

   From r_1, it reveals that the structures of the secret codes, which are $b_p ? b_q$ and $b_p b_q ?$, are preserved after mapping. Note that $b_p z_1 b_q$ and $b_p b_q z_2$ should be distinct to reserve the property of one-to-one mapping. On the one hand, we can achieve this by assigning the symbols of z_1 and z_2 carefully while mapping is conducted. On the other hand, there should be two symbols left for the assignments of z_1 and z_2 once b_p and b_q have been fixed during the mapping. The proof is therefore correct if $h \geq 5$. The proof of $C_{[0,1]} \leq C_{[0,0]}$ is finished now. Afterwards, another structural reduction, r_2, is defined as

$$r_2 : b_k b_p b_q \mapsto z_1 b_p b_q, \text{where } b_p, b_q \in B' = B - \{b_k\} \text{ and } z_1 \in B' - \{b_p, b_q\} .$$

   There should be one symbol left for the assignment of z_1 once b_p and b_q have been assigned. Hence, the proof is right if $h \geq 4$. In other words, $C_{[1,0]} \leq C_{[0,0]}$. From the results of r_1 and r_2, we know that $C_{[0,0]}$ dominates $C_{[0,1]}$ and $C_{[1,0]}$ when $h \geq 5$. This completes the proof of Lemma 1. □

3. $a_i b_j b_k, b_j a_i b_k,$ and $b_j b_k a_i$
   The guesses of this type are composed of a symbol in A and two symbols in B. These guesses can also be further classified into three kinds of guesses, i.e.,

$a_ib_jb_k, b_ja_ib_k,$ and $b_jb_ka_i$. Without loss of generality, $g = a_ib_jb_k$ is choosen to undertake the following discussions. Besides, the analyses of $b_ja_ib_k$ and $b_jb_ka_i$ can be derived in a similar way and so, they are ommited here. There are six nonempty substates after the codebreaker makes the guess, g. They are $C_{[2,0]}, C_{[1,1]}, C_{[0,2]}, C_{[1,0]}, C_{[0,1]},$ and $C_{[0,0]}$, respectively. Note that their corresponding cardinality are $(h-2), 2(h-2), (h-2), 2(h-2)(h-3), 4(h-2)(h-3),$ and $(h-2)(h-3)(h-4)$. Now, we show that $C_{[0,0]}$ dominates the other five substates if $h \geq 8$.

Lemma 2. *If the codebreaker encounters C^*, and then makes the guess, $g = a_ib_jb_k, b_ja_ib_k,$ or $b_jb_ka_i$, where $a_i \in A$ and $b_j, b_k \in B$, then $C_{[0,0]}$ dominates $C_{[0,1]}$, $C_{[1,0]}$, $C_{[0,2]}$, $C_{[1,1]}$, and $C_{[2,0]}$ when $h \geq 8$.*

Proof. Five structural reductions, called $r_3, r_4, r_5, r_6,$ and r_7, are defined as follows to certify that $C_{[0,1]} \leq C_{[0,0]}$, $C_{[1,0]} \leq C_{[0,0]}$, $C_{[0,2]} \leq C_{[0,1]}$, $C_{[1,1]} \leq C_{[1,0]}$, and $C_{[2,0]} \leq C_{[1,0]}$ respectively.

$$r_3 : \begin{cases} b_jb_pb_q \mapsto z_1b_pb_q \\ b_pb_qb_j \mapsto b_pb_qz_2, \text{ where } b_p, b_q \in B^{'} = B - \{b_j, b_k\}, \\ b_kb_pb_q \mapsto z_3b_pb_q \quad \text{and } z_1, z_2, z_3, z_4 \in B^{'} - \{b_p, b_q\}. \\ b_pb_kb_q \mapsto b_pz_4b_q \end{cases}$$

$$r_4 : \begin{cases} b_pb_jb_q \mapsto b_pz_1b_q, \text{ where } b_p, b_q \in B^{'} = B - \{b_j, b_k\} \\ b_pb_qb_k \mapsto b_pb_qz_2 \quad \text{and } z_1, z_2 \in B^{'} - \{b_p, b_q\}. \end{cases}$$

$$r_5 : \begin{cases} b_jb_kb_p \mapsto b_jz_1b_p \\ b_pb_kb_j \mapsto b_pz_2b_j, \text{ where } b_p \in B^{'} = B - \{b_j, b_k\} \\ b_kb_pb_j \mapsto b_kb_pz_3 \quad \text{and } z_1, z_2, z_3 \in B^{'} - \{b_p\}. \end{cases}$$

$$r_6 : \begin{cases} b_kb_jb_p \mapsto z_1b_jb_p, \text{ where } b_p \in B^{'} = B - \{b_j, b_k\} \\ b_jb_pb_k \mapsto z_2b_pb_k \quad \text{and } z_1, z_2 \in B^{'} - \{b_p\}. \end{cases}$$

$$r_7 : b_pb_jb_k \mapsto b_pb_jz_1, \text{ where } b_p \in B^{'} = B - \{b_j, b_k\} \text{ and } z_1 \in B^{'} - \{b_p\}.$$

Note that $z_1b_pb_q$, $b_pb_qz_2$, $z_3b_pb_q$, and $b_pz_4b_q$ in r_3 should be distinct to reserve the one-to-one mapping property. Likewise, $b_pz_1b_q$ and $b_pb_qz_2$ in r_4 should be distinct and $b_jz_1b_p$, $b_pz_2b_j$, and $b_kb_pz_3$ in r_5 should also be distinct while $z_1b_jb_p$ and $z_2b_pb_k$ in r_6 have to be distinct as well. We can attain this with assigning these symbols of z_1, z_2, z_3, and z_4 carefully when mapping is undertaken. In order to meet requirements of the assignments of z_i in $r_3, r_4, r_5, r_6,$ and r_7, the following conditions should be maintained respectively: $h \geq 8, h \geq 6, h \geq 6, h \geq 5,$ and $h \geq 4$. Consequently, it is true that $C_{[0,0]}$ dominates $C_{[0,1]}$, $C_{[1,0]}$, $C_{[0,2]}$, $C_{[1,1]}$, and $C_{[2,0]}$ while $h \geq 8$. Hence, the proof of Lemma 2 is completed. □

4. $b_ib_jb_k$
All symbols of this kind of guesses belong to B entirely. There are totally nine nonempty substates, which are $C_{[3,0]}$, $C_{[1,2]}$, $C_{[0,3]}$, $C_{[2,0]}$, $C_{[1,1]}$, $C_{[0,2]}$,

$C_{[1,0]}$, $C_{[0,1]}$, and $C_{[0,0]}$ respectively, as the codebreaker makes the guess, $g = b_i b_j b_k$. Notice that their cardinality are $1, 3, 2, 3(h-3), 6(h-3), 9(h-3), 3(h-3)(h-4), 6(h-3)(h-4)$, and $(h-3)(h-4)(h-5)$, respectively. In the following statements, we would certify that $C_{[0,1]} \le C_{[0,0]}$, $C_{[1,0]} \le C_{[0,0]}$, $C_{[0,2]} \le C_{[0,1]}$, $C_{[1,1]} \le C_{[1,0]}$, $C_{[2,0]} \le C_{[1,0]}$, $C_{[0,3]} \le C_{[0,0]}$, $C_{[1,2]} \le C_{[0,0]}$, and $C_{[3,0]} \le C_{[0,0]}$.

Lemma 3. *As the codebreaker encounters C^*, and then makes the guess, $g = b_i b_j b_k$, where $b_i, b_j, b_k \in B$, then $C_{[0,0]}$ dominates $C_{[0,1]}$, $C_{[1,0]}$, $C_{[0,2]}$, $C_{[1,1]}$, $C_{[2,0]}$, $C_{[0,3]}$, $C_{[1,2]}$, and $C_{[3,0]}$ when $h \ge 11$.*

Proof. Since the cardinalities of $C_{[3,0]}$, $C_{[1,2]}$, and $C_{[0,3]}$ are fixed numbers, then $C_{[0,0]}$ trivially dominates $C_{[3,0]}$, $C_{[1,2]}$, and $C_{[0,3]}$ as long as there are at least three symbols in B and thus, the three symbols can be permuted appropriately to map the three substates.

Below five definitions of structural reductions, which are named as r_8, r_9, r_{10}, r_{11}, and r_{12}, are provided as follows to confirm that $C_{[0,1]} \le C_{[0,0]}$, $C_{[1,0]} \le C_{[0,0]}$, $C_{[0,2]} \le C_{[0,1]}$, $C_{[1,1]} \le C_{[1,0]}$, and $C_{[2,0]} \le C_{[1,0]}$ respectively.

$$r_8 : \begin{cases} b_p b_i b_q \mapsto b_p z_1 b_q \\ b_p b_q b_i \mapsto b_p b_q z_2 \\ b_j b_p b_q \mapsto z_3 b_p b_q, \text{ where } b_p, b_q \in B^{'} = B - \{b_i, b_j, b_k\} \\ b_p b_q b_j \mapsto b_p b_q z_4 \quad \text{and } z_1, z_2, z_3, z_4, z_5, z_6 \in B^{'} - \{b_p, b_q\}. \\ b_k b_p b_q \mapsto z_5 b_p b_q \\ b_p b_k b_q \mapsto b_p z_6 b_q \end{cases}$$

$$r_9 : \begin{cases} b_i b_p b_q \mapsto z_1 b_p b_q \\ b_p b_j b_q \mapsto b_p z_2 b_q, \text{ where } b_p, b_q \in B^{'} = B - \{b_i, b_j, b_k\} \\ b_p b_q b_k \mapsto b_p b_q z_3 \quad \text{and } z_1, z_2, z_3 \in B^{'} - \{b_p, b_q\}. \end{cases}$$

$$r_{10} : \begin{cases} b_j b_i b_p \mapsto z_1 b_i b_p \\ b_p b_i b_j \mapsto b_p z_1 b_j \\ b_j b_p b_i \mapsto z_1 b_p b_i \\ b_j b_k b_p \mapsto b_j z_1 b_p \\ b_p b_k b_j \mapsto b_p b_k z_1, \text{ where } b_p \in B^{'} = B - \{b_i, b_j, b_k\} \\ b_k b_p b_j \mapsto b_k b_p z_1 \quad \text{and } z_1, z_2 \in B^{'} - \{b_p\}. \\ b_k b_i b_p \mapsto z_2 b_i b_p \\ b_k b_p b_i \mapsto b_k b_p z_2 \\ b_p b_k b_i \mapsto b_p z_2 b_i \end{cases}$$

$$r_{11} : \begin{cases} b_i b_p b_j \mapsto b_i b_p z_1 \\ b_i b_k b_p \mapsto b_i z_2 b_p \\ b_p b_j b_i \mapsto b_p b_j z_1, \text{ where } b_p \in B^{'} = B - \{b_i, b_j, b_k\} \\ b_k b_j b_p \mapsto z_2 b_j b_p \quad \text{and } z_1, z_2 \in B^{'} - \{b_p\}. \\ b_p b_i b_k \mapsto b_p z_1 b_k \\ b_j b_p b_k \mapsto z_2 b_p b_k \end{cases}$$

$$r_{12}: \begin{cases} b_ib_jb_p \mapsto b_iz_1b_p \\ b_ib_pb_k \mapsto z_1b_pb_k, \text{ where } b_p \in B' = B - \{b_i, b_j, b_k\} \\ b_pb_jb_k \mapsto b_pb_jz_1 \quad \text{and } z_1 \in B' - \{b_p\} . \end{cases}$$

Note that each secret code in each structural reduction, i.e., r_8, r_9, r_{10}, r_{11}, and r_{12}, should be distinct from each other to reserve the one-to-one mapping property. This can be attained by assigning these symbols of z_1, z_2, z_3, z_4, z_5, and z_6 carefully. To satisfy each assignment of z_i in r_8, r_9, r_{10}, r_{11}, and r_{12}, the following constraints have to be kept in correspondence with the order given: $h \geq 11, h \geq 8, h \geq 6, h \geq 6$, and $h \geq 5$. So, it is therefore correct that $C_{[0,0]}$ dominates $C_{[0,1]}, C_{[1,0]}, C_{[0,2]}, C_{[1,1]}, C_{[2,0]}, C_{[0,3]}, C_{[1,2]}$, and $C_{[3,0]}$ when $h \geq 11$. Hence, the proof of Lemma 3 is completed. □

After four kinds of guesses for the codebreaker are discussed, only three kinds of guesses among them are useful since the first one causes non-optimal results trivially. In order to simplify the notations, let $C^{(2)}$, $C^{(3)}$, and $C^{(4)}$ denote the hardest states caused by guesses of Type 2, Type 3, and Type 4, respectively. Hence, the difficulties of these three states have to be determined to choose the best guess for the codebreaker. The following lemma therefore describes the phenomena.

Lemma 4. *When the codebreaker encounters C^*, the hardest states caused by guesses of Type 2, Type 3, and Type 4, i.e., $C^{(2)}$, $C^{(3)}$, and $C^{(4)}$, are produced. Thus, we have $C^{(4)} \leq C^{(3)} \leq C^{(2)}$.*

Proof. From the meanings of $C^{(2)}$, $C^{(3)}$, and $C^{(4)}$, it reveals that $C^{(2)}$ is composed of secret codes that are permutations of $(h-1)$ symbols, and $C^{(3)}$ consists of what are permutations of $(h-2)$ symbols while the codes in $C^{(4)}$ are permutations of $(h-3)$ symbols. Let $S^{(2)}$, $S^{(3)}$, and $S^{(4)}$ denote the sets of symbols appearing in $C^{(2)}$, $C^{(3)}$, and $C^{(4)}$, respectively. Then, let the symbols in $S^{(2)}$, $S^{(3)}$, and $S^{(4)}$ be sorted separately according to the lexicographical order. A mapping is generated naturally if we map each symbol in $S^{(4)}$ to that in $S^{(3)}$ one by one in sorted order. So does the mapping between $S^{(3)}$ and $S^{(2)}$. Obviously, we have $C^{(4)} \leq C^{(3)} \leq C^{(2)}$. This completes the proof. □

After Lemma 1, Lemma 2, Lemma 3, and Lemma 4, we may conclude with the following lemma.

Lemma 5. *For a special state, C^*, which also represents a $3 \times h$ AB game $(11 \leq h \leq n)$, the optimal guess for the codebreaker now is $b_ib_jb_k$, where b_i, b_j, $b_k \in B$.*

Proof. From Lemma 4, $C^{(4)}$ is the easiest state to identify a secret code compared to $C^{(2)}$ and $C^{(3)}$. The goal of the codebreaker is to minimize the number of guesses required and so, the codebreaker has to choose the guess which results in $C^{(4)}$ in the worst situation. The optimal guess for the codebreaker is therefore $b_ib_jb_k$. □

5 The Devil's Strategy for the Codemaker

Since the mission of the codebreaker aims to minimize the number of guesses to acquire a secret code, the codemaker tries to maximize the number of guesses for the codebreaker if he decides to implement a devil's strategy. Hence, the worst case for the codebreaker means that his opponent conducts a devil's strategy (or called a worst response for the codebreaker) in each turn during the gaming process in order to maximize the number of guesses. In the follow-up, a lemma is exhibited to demonstrate what is the worst response for the codebreaker if he encounters a $3 \times h$ AB game, where $h \leq n$.

Lemma 6. *For a $3 \times h$ AB game, where $11 \leq h \leq n$, the codebreaker will require a maximum number of guesses to obtain the code while the codemaker answers* [0, 0] *after the codebreaker's guess.*

Proof. From Lemma 5, it is obvious that the codebreaker must choose $b_i b_j b_k$ as a guess for a $3 \times h$ AB game. After the codebreaker takes the optimal guess, nine substates will be formed. These substates are $C_{[0,0]}$, $C_{[0,1]}$, $C_{[1,0]}$, $C_{[0,2]}$, $C_{[1,1]}$, $C_{[2,0]}$, $C_{[0,3]}$, $C_{[1,2]}$, and $C_{[3,0]}$, respectively. $C_{[0,0]}$ dominates $C_{[0,1]}$, $C_{[1,0]}$, $C_{[0,2]}$, $C_{[1,1]}$, $C_{[2,0]}$, $C_{[0,3]}$, $C_{[1,2]}$, and $C_{[3,0]}$ in accordance with the result of Lemma 3. In other words, $C_{[0,0]}$ is the hardest substate among the nine ones. Conclusively, the codemaker must response [0, 0] as his worst response and this will result in the worst case for the codebreaker because of the maximum number of guesses. The proof is thus finished. □

6 Conclusions

From the above discussions, the optimal guess for the codebreaker and the adversary response for the codemaker, which refers to the worst case for the codebreaker as well, are eventually obtained with the consideration of the special state C^*. In the follow-up, all results mentioned above will be concluded to derive a theorem.

Theorem 1. *For a $3 \times n$ AB game, the minimum number of guesses for the codebreaker in the worst case is*

$$\begin{cases} \lfloor n/3 \rfloor + 3, & \text{if } 3 \leq n \leq 7 \\ \lfloor (n+1)/3 \rfloor + 3, & \text{if } n \geq 8. \end{cases}$$

Proof. At the beginning of a $3 \times n$ AB game, the n symbols are not used and then all secret codes are all equivalent. As a result, a secret code is chosen randomly as the first guess for the codebreaker. Nine substates are therefore produced and [0, 0] is taken as an adversary response according to Lemma 6. Afterwards, $C_{[0,0]}$, which results from the first response, matches the attribution of the special state C^* described in Lemma 5. Thus, Lemma 5 can be applied to this state. We find that the situations mentioned in Lemma 5 and Lemma 6 will appear alternately in the following gaming process. So we have the following recurrence.

$$T(n) = T(n-3) + 1, \text{ when } n > 11. \tag{1}$$

The minimum number of guesses cannot be obtained easily with the use of analyses when $n \leq 11$ because of the irregular behavior. So, a refined exhaustive search, which originates from Huang *et al.* [9], is adopted to acquire the results. After the use of computer programs written with this approach, the minimum numbers of guesses required for the codebreaker in the worst case are obtained in several hours and they are 4, 4, 4, 5, 5, 6, 6, 6, and 7, respectively when $n = 3, 4, 5, 6, 7, 8, 9, 10,$ and 11. For example, an optimal strategy for 3×7 AB game is considered with $S = \{0, 1, 2, 3, 4, 5, 6\}$. If the codemaker takes 165 as a secret code, the gaming process will be as follows: 012, $[0, 1]$, 023, $[0, 0]$, 041, $[0, 1]$, 156, $[1, 2]$, 165, $[3, 0]$. In other words, the codebreaker requires 5 guesses to identify 165 while playing the worst-case optimal strategy.

We derive the above recurrence (1) and conclude with the results of smaller values of n. Hence, the closed form of the formula is exhibited as follows.

$$\begin{cases} \lfloor n/3 \rfloor + 3, & \text{if } 3 \leq n \leq 7 \\ \lfloor (n+1)/3 \rfloor + 3, & \text{if } n \geq 8. \end{cases}$$

This completes the proof. □

Partial results of $3 \times n$ AB games, $3 \leq n \leq 16$, are summarized in Table 1. As $3 \times n$ AB games have been solved successfully, a natural generalization is to explore the techniques for $m \times n$ AB games, where $m \geq 4$. This problem remains open. We hope that the methods proposed here could help other related research in the future.

Table 1. The minimum number of guesses for $3 \times n$ AB games in the worst case

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# of guesses	4	4	4	5	5	6	6	6	7	7	7	8	8	8

Acknowledgements

We thank anonymous referees for their valuable comments. This research was supported in part by a grant NSC 97-2221-E-003-008 from National Science Council, ROC.

References

1. Merelo-Guervos, J.J., Castillo, P., Rivas, V.M.: Finding a needle in a haystack using hints and evolutionary computation: the case of evolutionary MasterMind. Applied Soft Computing 6(2), 170–179 (2006)
2. Knuth, D.E.: The computer as Mastermind. Journal of Recreational Mathematics 9, 1–6 (1976)
3. Irving, R.W.: Towards an optimum Mastermind strategy. Journal of Recreational Mathematics 11(2), 81–87 (1978)

4. Neuwirth, E.: Some strategies for Mastermind. Mathematical Methods of Operations Research 26, 257–278 (1982)
5. Norvig, P.: Playing Mastermind optimally. ACM SIGART Bulletin 90, 33–34 (1984)
6. Koyama, K., Lai, T.W.: An optimal Mastermind strategy. Journal of Recreational Mathematics 25, 251–256 (1993)
7. Rosu, R.: Mastermind. Master's thesis, North Carolina State University, Raleigh, North Carolina (1999)
8. Barteld, K.: Yet another Mastermind strategy. ICGA Journal 28(1), 13–20 (2005)
9. Huang, L.T., Chen, S.T., Huang, S.J., Lin, S.S.: An efficient approach to solve Mastermind optimally. ICGA Journal 30(3), 143–149 (2007)
10. Shapiro, E.: Playing Mastermind logically. ACM SIGART Bulletin 85, 28–29 (1983)
11. Chen, S.T., Lin, S.S., Huang, L.T., Hsu, S.H.: Strategy optimization for deductive games. European Journal of Operational Research 183(2), 757–766 (2007)
12. Kalisker, T., Camens, D.: Solving Mastermind using genetic algorithms. In: Cantú-Paz, E., Foster, J.A., Deb, K., Davis, L., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Kendall, G., Wilson, S.W., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A., Dowsland, K.A., Jonoska, N., Miller, J., Standish, R.K. (eds.) GECCO 2003. LNCS, vol. 2724, pp. 1590–1591. Springer, Heidelberg (2003)
13. Singley, A.: Heuristic solution methods for the 1-dimensional and 2-dimensional Mastermind problem. Master's thesis, University of Florida (2005)
14. Chen, S.T., Lin, S.S., Huang, L.T.: A two-phase optimization algorithm for Mastermind. The Computer Journal 50(4), 435–443 (2007)
15. Berghman, L., Goossens, D., Leus, R.: Efficient solutions for Mastermind using genetic algorithms. Computers and Operations Research 36(6), 1880–1885 (2009)
16. Chen, S.T., Lin, S.S.: Optimal algorithms for $2 \times n$ AB games - a graph-partition approach. Journal of Information Science and Engineering 20(1), 105–126 (2004)
17. Chen, S.T., Lin, S.S.: Optimal algorithms for $2 \times n$ Mastermind games - a graph-partition approach. The Computer Journal 47(5), 602–611 (2004)
18. Goddard, W.: Mastermind revisited. Journal of Combinatorial Mathematics and Combinatorial Computing 51, 215–220 (2004)

Automated Discovery of Search-Extension Features

Pálmi Skowronski[1], Yngvi Björnsson[1], and Mark H.M. Winands[2]

[1] Reykjavík University, School of Computer Science,
Menntavegi 1, Reykjavík 101, Icleand
{palmis01,yngvi}@ru.is

[2] Games and AI Group, Department of Knowledge Engineering,
Maastricht University, Maastricht, The Netherlands
m.winands@maastrichtuniversity.nl

Abstract. One of the main challenges with selective search extensions is designing effective move categories (features). Usually, it is a manual trial-and-error task, which requires both intuition and expert human knowledge. Automating this task potentially enables the discovery of both more complex and more effective move categories. The current work introduces *Gradual Focus*, an algorithm for automatically discovering interesting move categories for selective search extensions. The algorithm iteratively creates new more refined move categories by combining features from an atomic feature set. Empirical data is presented for the game Breakthrough showing that Gradual Focus looks at a number of combinations that is two orders of magnitude fewer than a brute-force method does, while preserving adequate precision and recall.

1 Introduction

The $\alpha\beta$ algorithm is one of the fundamental and most effective search techniques used by game-playing programs for playing two-person adversary board games, such as chess and checkers. Over the years many enhancements have been proposed to improve its efficiency. For instance, we know that the standard strategy of exploring all alternatives to the same fixed depth is not most effective. Instead, various techniques have been proposed for searching the game tree more selectively, where some lines of play are terminated prematurely whereas others are explored more deeply. The former scenario is referred to as search reductions (or speculative pruning) and the latter as search extensions. In chess, for example, it is common to resolve forced situations, such as checks and recaptures, by searching them more deeply.

The move-decision quality of the alpha-beta algorithm is greatly influenced by the choices of which lines are investigated deeply [1,2]. Therefore, the design of an effective search-extension scheme is fundamental to any high-performance $\alpha\beta$-based game-playing program. The typical approach for incorporating search extensions into a game-playing program is to predefine a set of move categories (e.g., checks and recaptures), and then associate a different cost weight to each

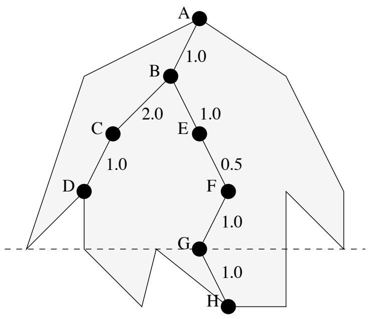


Fig. 1. Fractional-ply example

category. During the search, each move is categorized as belonging to one of the predefined move classes, and the depth of the current search path then becomes the sum of the weights of the moves on the path. If all move categories have the same weight, one would obtain the regular behaviour of a fixed-depth search. However, by assigning a weight of less than one to selected move categories, e.g., checking moves, such lines of play will be explored more deeply. This scheme is commonly referred to as fractional-ply extensions [3,4], depicted in Figure 1.

The weight of each move category, i.e., its fractional-ply value, is either manually assigned a value based on trial and error, or, alternatively, automatically tuned from game records [5,6], test-suites [7], or during play [7,8,9]. For creating the move categories the standard practice is to do it manually based on intuition and domain expertise. In this work we investigate ways for automatically discovering useful move categories for use in game-playing programs. The main contribution is a new method for automatically discovering such features, called *Gradual Focus*. We experiment with it in the game Breakthrough, where there exists little knowledge of what comprises good moves to extend on.

The paper is structured as follows. In Section 2 we give an overview of relevant background material, followed by a description of the new Gradual Focus feature-discovery algorithm in Section 3. The algorithm is empirically evaluated in Section 4, and finally we conclude and discuss future work in Section 5.

2 Background

The general approach to automated feature discovery is to start with a set of so-called *atomic* features. The atomic features are typically simple features expressing trivial facts about the problem domain, for example, type and placement of pieces. As a standalone, these features are not necessary effective, for example, because they might be too general. More sophisticated features are then constructed by combining the atomic features in various ways, e.g., by using the logical operators $\wedge$ and $\vee$. This may be done in an iterative fashion, that is, first pair-wise combinations are created, then three-wise, etc. The problem ,though, is that the number of possible feature combinations grows exponentially in each iteration. For example, a brute-force power set method would generate in total $2^n - 1$ features from an atomic feature set of size n. Consequently, to limit the growth rate, a selective mechanism judging the merits of newly create features may be applied, carefully choosing which features to evolve further.

In the context of game-playing, automatic feature discovery has first and foremost been applied to the learning of evaluation functions, as opposed to search-control features. One of the first such approaches was introduced in the system ZENITH [10]. The system works in a way backwards to the general approach described above: it starts with a single feature, a logical formula describing the goal of the game. It then gradually breaks the goal down into simpler sub-goals by using predefined generic actions in the form of decomposition, abstraction, goal regression, and specialization. Logical features of this kind have also been successfully used to extract patterns that can be used as features for general game playing [11]. In contrast, the system GLEM [12], creates new features by gradually combining mutually exclusive atomic features along the lines described above. The method is additionally capable of learning an importance weight for each of the newly created features. This method was used to construct a high-quality evaluation function for the Othello program LOGISTELLO [13], although, in that case, the features were provided manually and GLEM used only for tuning their relative weights. A different feature-combination approach was used to learn an evaluation function for a program to play the card game Hearts [14]. All possible pair-wise, three-wise, and four-wise combinations were created in more or less a brute-force manner and a reinforcement-learning approach then used to learn their relative importance. Finally, learning of move-patterns for a plausible move generation in chess is presented in [15].

3 Gradual Focus

Gradual focus (GF), the method we introduce here, is a more intelligent way of constructing interesting features than an exhaustive power-set method. GF combines atomic features in an iterative fashion, where each iteration creates a set of more refined features, gradually narrowing their focus, using a variety of pruning methods to reduce the number of possible combinations.

3.1 GF Overview

Given a set of atomic features GF combines these features using an $\wedge$ operator. The features are combined one level at a time, i.e., one-wise, two-wise, three-wise, etc., and their quality evaluated. Those features that do not show an improvement are pruned off and prevented from occurring as subsets in later feature combinations. This process is repeated until no more combinations are formed.

Figure 2 shows an example of this process. The first level consists of the atomic features, called *Base* set, and the second level shows the feature set generated by the first iteration, where all two-wise combinations of the *Base* set are created. Each feature in the set is evaluated individually and those that perform

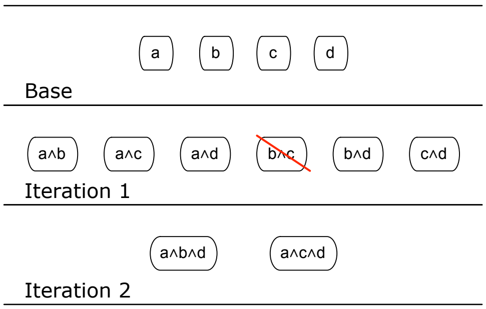


Fig. 2. Overview of Gradual Focus

worse than either parent are discarded, as feature $b \wedge c$ in this example. The next iteration evolves the surviving features further by combing the two-wise features with the features in the *Base* set. As the $b \wedge c$ feature has been discarded it is not evolved any further, nor are any evolving features containing $b \wedge c$ allowed. This results in only two three-wise combinations. Four-wise combinations cannot be formed either as $b \wedge c$ is forbidden as a subfeature, so the process halts.

3.2 Implementation Details

A pseudo-code for the GF algorithm is shown as Algorithm 1. The variable *BlackList* maintains disproven features, while the *Output* variable maintains the viable features. The *Neutral* variable is the empty feature; it is evaluated in *line 3* to establish a baseline of how the tree search behaves without search extensions. The function *evaluateFitness*, which will be discussed in detail in Section 3.6, returns a numerical value indicating the quality, i.e., the fitness of a feature. Each of *Base's* features is also evaluated *(line 5)*. Those features that perform below an expected level of quality *(lines 7-13)* can optionally be removed from the *Base* set, reducing GF's branching factor. This pruning method, called *threshold pruning*, is discussed in a more detail in Section 3.5.

The iterative feature-evolution process is done in *lines 17-26*. The function *evolveFeatures (line 19)* evolves the features in the *workSet* by combining them with the *Base* set, and using the *BlackList* to remove disallowed combinations. Both the *Base* and the *workSet* sets must be sorted in a descending order by evaluation score *(lines 14 and 18)* for *evolveFeatures* to work as intended.

The assessment of newly evolved features occurs in *line 20-25*, where each of the newly evolved features is evaluated *(line 22)* and its evolutionary direction assessed within *filterFeature (line 23)* as either ascending or descending. Descending combinations are added to the *BlackList*, thereby removing them and their descendants from the evolutionary process, while ascending combinations are returned from the function and added to GF's output *(line 23)*. Disproving a feature within *filterFeature* might also disprove other features in the *workSet* that are yet to be evaluated, which is why features must be compared against

Algorithm 1. *featureLearner(ref Base)*

```
 1: $BlackList \leftarrow \{\}$
 2: $Output \leftarrow \{\}$
 3: evaluateFitness( Neutral )
 4: for all $b \in Base$ do
 5:     evaluateFitness( b )
 6: end for
 7: if UseThreasholdPruning then
 8:     for all $b \in Base$ do
 9:         if $b_{value} < \delta$ then
10:             $Base \leftarrow Base \setminus \{b\}$
11:         end if
12:     end for
13: end if
14: sortDesc( Base )
15: $Output \leftarrow Neutral \cup Base$
16: $workSet \leftarrow Base$
17: while $workSet \neq \{\}$ do
18:     sortDesc( workSet )
19:     $workSet \leftarrow$ evolveFeatures( workSet, Base, BlackList )
20:     for all $f \in workSet$ do
21:         if isCompositionAllowed( f, BlackList ) then
22:             evaluateFitness( f )
23:             $Output \leftarrow Output \cup$ filterFeature( f, BlackList )
24:         end if
25:     end for
26: end while
27: sortDesc( Output )
28: display( Output )
```

the *BlackList* each time in the loop before they are evaluated *(line 21)*. This evolution process is iterated until no new features can be formed, the *evolveFeatures* function returns an empty set *(line 19 and 17)*, meaning that all promising evolution paths have been explored.

The implementation of the *evolveFeatures* routine is shown as Algorithm 2, where A is a set of features to be evolved and B is the set of features to combine with. The same feature can be composed in various ways with different first and second parent (i.e., $a \wedge b$ and $b \wedge a$ are considered the same feature). However, to simplify the evaluation of features we impose an order such that the first parent must have a greater fitness value than the second parent, which is why the input sets are sorted in a descending order before the evolution phase and then paired together from left to right. To prevent unnecessary combinations, the features can neither belong to the same group nor can a subset of the new feature c be on the *BlackList (line 4 and 6)*. Moreover, duplicate combinations are not allowed *(line 6)*. In the Sections 3.3, 3.4, and 3.5, we discuss three enhancements. Feature groups prevent illogical combinations and will be discussed in a more detail in

Algorithm 2. *evolveFeatures(A, B, BlackList)*

```
1: $new \leftarrow \{\}$
2: for all $a \in A$ do
3:   for all $b \in B$ do
4:     if $\neg$ belongToSameGroup( a, b) then
5:       $c \leftarrow$ combine( a, b )
6:       if isCompositionAllowed( c, BlackList ) $\wedge$ $c \notin new$ then
7:         $new \leftarrow new \cup c$
8:       end if
9:     end if
10:  end for
11: end for
12: return new
```

Section 3.3. Combinations that are not pruned away are added to a new set *(line 7)* which is then returned *(line 12)* as a new generation of features.

The assessment for the evolution progress is shown in Algorithm 3. In *line 1* the new feature's value is compared against its first parent. The value of ϵ is domain-specific and is added to the feature's value to control the level of improvement needed for the feature to evolve further. Features that do not improve upon their parent are pruned (Lineair Tree Pruning) by adding them to the *BlackList (line 2)*, thus preventing them from occurring in future evolution sets. Features that surpass their parent, are returned *(line 11)*. Lines 3-8 in the algorithm are a part of the *linear tree pruning* method to be discussed in Section 3.4. Thereafter Section 3.5 discusses threshold pruning.

3.3 Groups

Some features are not compatible in a sense that combining them is meaningless in the context of the game at hand, e.g., in chess: *capture the king* or *move a pawn and rook*. Features can thus be put in advance into logical groups, where

Algorithm 3. *filterFeature(newFeature, ref BlackList)*

```
1: if $newFeature_{value} < newFeature_{firstParentValue} + \epsilon$ then
2:   $BlackList \leftarrow BlackList \cup newFeature$
3:   if UseLinearTreePruning then
4:     for all $child \in$ getChildren( $newFeature_{secondParent}$ ) do
5:       $banned \leftarrow$ combine( $newFeature_{firstParent}$, child )
6:       $BlackList \leftarrow BlackList \cup banned$
7:     end for
8:   end if
9:   return {}
10: else
11:   return newFeature
12: end if
```

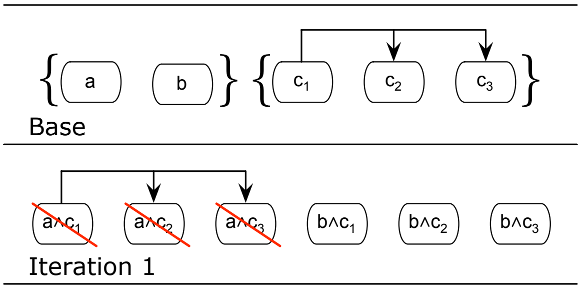


Fig. 3. Linear Tree Pruning example

combining features within a group is not allowed (*line 4* in Algorithm 2). This prevents many useless combinations from being formed. As an example, a set of six features divided into two equal groups, creates 63 different feature combinations without groups, but only 15 if groups are used. Combined features inherit the groups of their parents, making them belong to more than one group.

3.4 Linear Tree Pruning

Features within a group can form a natural hierarchy, where some features are subsets of others. This tree hierarchy can be used to predict the outcome of future evaluations and prune off those that are expected to be inferior. The logic is that if a new combination formed with a parent feature is pruned, then other combinations with that feature's children can also be eliminated.

Figure 3 shows an example of linear tree pruning. The *Base* set has been divided into two groups, and the second group has a hierarchy with c_2 and c_3 being children of c_1. In the first iteration GF creates all two-wise combinations which are then evaluated. Assuming the combination $a \wedge c_1$ is pruned, then if linear tree pruning is used the features $a \wedge c_2$ and $a \wedge c_3$ would also be pruned.

This method cannot guarantee that interesting combinations would not be pruned. The reason is that combinations with a highly frequent parent feature might extend too aggressively, whereas a combination with its less frequent children might not. The method is thus optional, as shown in Algorithm 3 in *lines 3-8*. It retrieves all the children of the second parent feature *(line 4)* and creates a new combination with the first parent and each of the retrieved children features *(line 5)*, which are then added to the *BlackList* *(line 6)*.

3.5 Threshold Pruning

Threshold pruning is an attractive optional pruning method available for GF. It removes all features from the *Base* set that are below a provided quality threshold, determining those features immediately as disadvantageous and therefore not eligible to participate in the evolution process (*lines 7-13* in Algorithm 1).

This is potentially a very effective pruning method as it reduces the exponential growth of the number of combined features at all iterative levels, but at the risk of wrongfully pruning away potentially good candidates. The risk can be lessened by marking which features can be safely pruned; this however requires knowledge of the search domain. A sensible choice for the threshold parameter δ *(line 9)* would be to approximate it around the value of the *Neutral* feature.

3.6 Fitness Evaluation

The true quality of a search-extension feature can be found only by using it in actual game-play. However, playing games is time-consuming, so instead we use a suite of selected test positions where the best move is known. Information about the feature's effect on the search is collected: number of solved positions, mean iteration depth, mean height, and feature's frequency in the search. Various methods to evaluate the feature's quality can be formed based on the gathered information, but we choose to use straightforwardly the number of solved positions (i.e., best move played) as it directly measures a feature's effectiveness.

4 Empirical Results

We empirically evaluate the GF method by discovering search-extension features for the game Breakthrough. The experiments were run on a Linux CentOS 5 Intel(R) Xeon (TM) 3.00GHz CPU machine with 2GB of memory.

4.1 Breakthrough

Breakthrough [16] is a two-person perfect-information game created by Dan Troyka in 2001. The game is played on a chess-board where each player has 16 pawn-like pieces that fill the two front and back rows of the board. The objective of the game is to break through the opponent's ranks and advance a piece to the opponents back rank. Despite its simple rules, which are given below, the game requires a sophisticated strategy to play at an expert level.

1. Players' two back rows are filled with their pieces at the start of the game.
2. Players choose which side starts (our program assumes that White starts).
3. Players alternate moving a piece.
   (a) One square forward or diagonally-forward to unoccupied squares.
   (b) One square diagonally-forward to a square containing an opponent's piece, capturing opponent's piece and removing it from the board.
4. Capture moves are not forced.
5. A game ends when a player's piece reaches the opponent's back rank.

Table 1. Description of Breakthrough's base features

Feature	Feature Description
Ud	Moved piece is not threatened on destination square.
PP	Piece's direction is unhindered towards opponent's back rank.
Rc	Capture previously moved piece.
C	Capture opponents piece.
Ms	Majority of squares surrounding moved piece is occupied by players pieces, forming a greater mass.
Rdb	Players half of the board.
RdBb	First and second rank of the board.
RdBt	Third and fourth rank of the board.
Rdt	Opponents half of the board.
RdTb	Fifth and sixth rank of the board.
RdTt	Seventh and eighth rank of the board.
Edg	The board's edges, columns *a, b, g,* and *h.*
Mr	The board's middle, columns *c, d, e,* and *f.*
Udp	Prepare to move around opponent's piece. Piece is not threatened and standing opposite opponent's piece.
Bv2	Block opponent's advancement by placing piece in front of it, creating a vertical defensive line of two pieces.

4.2 Experiments Setup

A description of the atomic features forming GF's *Base* set is shown in Table 1. Each feature belongs to its own group except the following larger groups: {*Rdb, RdBb, RdBt*}, {*Rdt, RdTb, RdTt*}, and {*Edg, Mr*}. Two of these have a tree-hierarchy: *RdBb* and *RdBt* are *Rdb's* children, and *RdTb* and *RdTt* are *Rdt's* children. The features are evaluated using a fixed fractional-ply value of 0.5, chosen somewhat arbitrary although such that it is neither too conservative nor too aggressive. Each feature was evaluated using our Breakthrough program searching 500,000 nodes per search, which corresponds to approximately 5-ply search. The program uses $\alpha\beta$ search with iterative deepening, and a simple but fairly effective heuristic based on material and bonuses for advanced non-attacked pieces. GF's evolution parameter ϵ is set to 3 to compensate for fluctuating evaluations, and the threshold pruning parameter δ equals to the *Neutral* feature's value. As there exists no standard position test-suite for Breakthrough we created a set of 302 positions which were picked from a game of self-play where the terminal state could be reached in 7 plies. This allowed us to identify unambiguously the best move, but has the drawback of all the positions being taken from the endgame. Three programs using different heuristic evaluations were used to obtain a variety of endgame positions.

4.3 Feature Evolution Results

Four different instantiations of GF were evaluated. The results are shown in Table 2 where *Default* disregards previously presented grouping of features, plac-

Table 2. Gradual focus evaluations in Breakthrough

Method type	Hours	Evaluations #	% of $\mathcal{P}$	Overlooked combinations
Power Set	-	32,768	100%	
Default	30.6	138	0.42%	
Default + G	24.9	112	0.34%	0
Default + G + LTP	23.7	105	0.32%	0
Default + G + LTP + TP	5.8	27	0.08%	5

ing each feature in a group of its own. The GF's enhancements, *Groups* (G), *Linear tree pruning* (LTP), and *Threshold pruning* (TP), are then cumulatively added using the previously described grouping of features. The calculated result of a power-set's feature expansion are also shown.

As can be seen by the *Default* instance, GF reduces the number of generated feature combinations immensely compared to the brute-force power-set method, and without overlooking any interesting combinations. Thirteen of these evaluations are incompatible combinations that can never occur in the game, which were prevented in the *G* instance. Adding *LTP* improves the pruning slightly further without overlooking any previously interesting combinations. Additionally, *TP* further reduces the number of evaluations substantially, but at the cost of overlooking five of *Default*'s top ten most interesting combinations, *Ud-PP-Rdt*, *Ud-Rdt*, *PP-Rdt*, *PP-Rdt-Edg*, and *Rc-Rdb*.

4.4 Precision and Recall

GF's findings were compared with the complete set of all one, two, and tree-wise combinations, in all 377 features. The resulting top 25 features are shown in Table 3 along with the number of positions they solve and their first parent. Features written in italics are less effective descendants of already discovered features and as such redundant as their benefits are already obtained by the use of their parent. Ignoring these features leaves only 11 of the original 25 features.

The top features that GF returns are exactly these 11 features. Thus, in this domain, GF offers both perfect precision (number of correctly identified features) and perfect recall (how large portion of the interesting features were discovered).

4.5 Tournament Results

The top-ten features suggested by GF as interesting were also evaluated through self-play. Table 4 shows the results, with the last two columns contrasting the feature frequency when (1) searching the test-suite and (2) in actual games.

All but three of the features lead to an improved play. The features *Rc-Rdb* and *PP-Rdt-Edg* have a little effect on the game, which can in part be explained by their low frequency. However, that alone is a not a sufficient explanation as *PP-RdTt* with even a lower frequency is doing well. That feature is extending

Table 3. Tree-Wise Combinations in Breakthrough

	Feature	Solved	Parent		Feature	Solved	Parent
★	Ud-RdTt	235	Ud		*PP-Edg-RdTt*	*115*	*PP-RdTt*
	Ud-PP-RdTt	*224*	*Ud-RdTt*		*Ud-Ms-RdTt*	*114*	*Ud-RdTt*
★	PP-RdTt	211	PP		*Ud-PP-RdTb*	*111*	*Ud-PP*
★	RdTt	202	–		*Ms-RdTt*	*110*	*RdTt*
★	Ud-PP-Rdt	173	Ud-PP		*Ud-PP-Mr*	*109*	*Ud-PP*
★	Ud-PP	155	Ud	★	PP-Rdt-Edg	108	PP-Rdt
	Ud-Edg-RdTt	*149*	*Ud-RdTt*		*Ud-Rdt-Mr*	*105*	*Ud-Rdt*
	Edg-RdTt	*142*	*RdTt*	★	Ud	102	–
	Ud-RdTt-Mr	*141*	*Ud-RdTt*		*Ud-Rdt-Edg*	*102*	*Ud-Rdt*
	PP-RdTt-Mr	*140*	*PP-RdTt*	★	PP-Rdt	98	PP
	RdTt-Mr	*138*	*RdTt*	★	PP-Edg	97	PP
★	Ud-Rdt	125	Ud	★	Rc-Rdb	96	Rc
	Ud-PP-Edg	*122*	*Ud-PP*				

Table 4. Features' result in Breakthrough

Feature	Games #	Wins %	Conf. Int.	Frequency Suite	Frequency Games
Ud-Rdt	2400	58.17%	±1.97	15.57%	4.71%
PP-RdTt	2400	57.04%	±1.98	2.87%	0.87%
RdTt	2400	56.54%	±1.98	8.75%	1.91%
Ud-RdTt	2400	55.96%	±1.99	7.76%	1.90%
Ud-PP	2400	53.92%	±1.99	8.44%	2.31%
Ud-PP-Rdt	2400	53.50%	±2.00	6.94%	1.99%
PP-Rdt	2400	52.03%	±2.00	12.33%	3.92%
Rc-Rdb	2400	49.29%	±2.00	1.55%	1.48%
PP-Rdt-Edg	2400	49.13%	±2.00	3.18%	1.18%
Ud	2400	46.46%	±2.00	32.18%	43.37%

only on safe pawn moves just about to reach the back rank and thus, even though infrequent, almost always results in a more accurate evaluation score. In contrast, the *Ud* feature has a negative effect on in-game performance, whereas it was slightly beneficial on the test-suite. We see that this type of extension is substantially more frequent in actual game play than in the test-suite, which might be enough to tilt the balance. This could probably be avoided by using a more diverse test-suite containing start, middle, and endgame positions.

5 Conclusions

We introduced a new method for learning search-extension features, called *Gradual Focus.* It iteratively creates new refined features by combining atomic features from a base set with the $\wedge$ operator, using various merit-based pruning techniques to select which features to elaborate. We evaluated the method in the game Breakthrough, where there exists little domain knowledge of what makes up good move categories to extend on. The method learned several promising search-extensions features from a suite of test positions. Moreover, it required several orders of magnitude less time than a brute-force approach while demonstrating both an excellent precision and recall. The learned features, when used in regular game play, significantly improved our program's playing strength. Also, the method does not require the learned features necessarily to be move categories, and GF could thus be used for other search-control features as well.

There is still much scope for improvements and we view this work as a first step in exploring automatic discovery of search-control features. In particular, currently we use the number of solved positions as the only indicator of a feature's quality. However, by also monitoring other statistics when evaluating a feature, such as its frequency in the search, one could pinpoint promising evolution paths more intelligently. For example, a feature that is much less frequent than another might be preferred even if it solves a slightly less number of test positions. A second issue is that we evaluate all features using a fixed FP value, which undeniably excludes the discovery of potentially useful features (we have observed that a feature's quality is quite sensitive to its FP value). Thus, combining learning of features with methods for learning FP values is a worthwhile avenue for future research. Finally, it would be interesting to explore the method in other game domains, and we have started preliminary work in chess.

Acknowledgments

This research was supported by grants from The Icelandic Centre for Research (RANNIS) and by a Marie Curie Fellowship of the European Community programme Structuring the ERA under contract MIRG-CT-2005-017284. We also thank the anonymous reviewers for their comments.

References

1. Anantharaman, T.S., Campbell, M.S., Hsu, F.: Singular extensions: adding selectivity to brute-force searching. Artificial Intelligence 43(1), 99–109 (1990)
2. Beal, D.F., Smith, M.C.: Quantification of search extension benefits. ICCA Journal 8(4), 205–218 (1995)
3. Hyatt, R.M.: Crafty. A chess program (1996) (March 27, 2008), `ftp://ftp.cis.uab.edu/pub/hyatt`
4. Levy, D., Broughton, D., Taylor, M.: The SEX algorithm in computer chess. ICCA Journal 12(1), 10–21 (1989)
5. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 146–153 (2002)
6. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)

7. Björnsson, Y.: Selective Depth-First Game-Tree Search. Phd dissertation, University of Alberta (2002)
8. Björnsson, Y., Marsland, T.A.: Learning extension parameters in game-tree search. Information Sciences 154(3-4), 95–118 (2003)
9. Kocsis, L., Szepesvári, C., Winands, M.H.M.: RSPSA: Enhanced Parameter Optimization in Games. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H.H.L.M(J.) (eds.) CG 2005. LNCS, vol. 4250, pp. 39–56. Springer, Heidelberg (2006)
10. Fawcett, T.E., Utgoff, P.E.: Automatic feature generation for problem solving systems. In: Intern. Conf. on Machine Learning (ICML), pp. 144–153 (1992)
11. Kaneko, T., Yamaguchi, K., Kawai, S.: Automated identification of patterns in evaluation functions. In: Advances in Computer Games, vol. 10, pp. 279–298 (2003)
12. Buro, M.: From simple features to sophisticated evaluation functions. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 126–145. Springer, Heidelberg (1999)
13. Buro, M.: Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In: Games in AI Research, pp. 77–96 (1999)
14. Sturtevant, N.R., White, A.M.: Feature construction for reinforcement learning in hearts. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 122–134. Springer, Heidelberg (2007)
15. Finkelstein, L., Markovitch, S.: Learning to play chess selectively by acquiring move patterns. ICCA Journal 21(2), 100–119 (1998)
16. Handscomb, K.: 8×8 game design competition: The winning game: Breakthrough ... and two other favorites. Abstract Games Magazine 7 (2001)

Deriving Concepts and Strategies from Chess Tablebases

Matej Guid, Martin Možina, Aleksander Sadikov, and Ivan Bratko

Artificial Intelligence Laboratory, Faculty of Computer and Information Science, University of Ljubljana, Slovenia

Abstract. Complete tablebases, indicating best moves for every position, exist for chess endgames. There is no doubt that tablebases contain a wealth of knowledge, however, mining for this knowledge, manually or automatically, proved as extremely difficult. Recently, we developed an approach that combines specialized minimax search with the argument-based machine learning (ABML) paradigm. In this paper, we put this approach to test in an attempt to elicit human-understandable knowledge from tablebases. Specifically, we semi-automatically synthesize knowledge from the KBNK tablebase for teaching the difficult king, bishop, and knight versus the lone king endgame.

1 Introduction

Chess tablebases [1] have enabled people a glimpse of how perfect play looks like. It seems, however, that people are ill adapted to understanding this perfection. While tablebases are a powerful help to computers, people are for the most part puzzled by the style of play generated by tablebases. Yet, people would like to learn as much as possible, and there is no doubt that tablebases contain an enormous amount of potential knowledge — but in a form not easily accessible to a human mind.

There have been many attempts to extract knowledge from tablebases. Perhaps two best documented examples are a research project carried out by the chess study specialist John Roycroft [2] and the work by grandmaster John Nunn resulting in two books on pawnless endings [3,4]. All the attempts, however, had their own, mostly limited success.

The goal of Roycroft's study was that he would learn himself reliably to play the KBBKN endgame (king and two bishops vs. king and knight). This endgame was for a long time considered generally to be drawn, until the KBBKN tablebase was computed. The tablebase showed that the side with two bishops can usually force a win, but the winning play is extremely difficult and takes a long sequence of moves under optimal play by both sides. Many moves in the optimal play for the stronger side are completely obscure to a human. Roycroft tried to extract a human-executable winning strategy by the help of this tablebase, trying to discover manually important concepts in this endgame which would enable a human to win reliably. After a one year's effort, the project ended with a rather limited success when Roycroft's accumulated skill for this endgame was still not quite sufficient actually to win against the tablebase in many of the won KBBKN positions.

The task of learning is not any easier for the computer. In an overview of learning methods in games, Fürnkranz indicated that machine learning of understandable and usable concepts over the years did not yield much success [5]. There have been various attempts to bridge the gap between perfect information stored in tablebases and human-usable strategies. While some of these approaches succeeded for relatively small domains (such as the KRK endgame in chess), the resulting models are hardly intelligible to human experts [6], not to mention beginners and novices. All related work did not result in breakthroughs in more complex domains. Moreover, the research questions (1) how to learn human-understandable models? and (2) how to use the models to generate instructions suitable for teaching humans? remained open.

In a way learning from tablebases resembles closely the extraction of an expert's tacit knowledge when constructing a knowledge base of an expert system — in both cases the knowledge is difficult to extract. Recently, we proposed a new paradigm, which facilitates semi-automatic elicitation of knowledge in the form of rules. We successfully applied it to creating a knowledge base of an expert system that recognizes bad bishops in chess middlegames and is able to explain its decisions [7,8].

The goal of this paper is a practical demonstration of the usefulness of our approach for semi-automatic elicitation of knowledge. We used a recently developed method within the aforementioned paradigm for learning strategic goal-based rules. We harvested the tablebase to extract useful concepts and strategies which the domain expert in close collaboration with the machine learning tool turned into a textbook (and computer aid) for teaching the difficult-to-master KBNK (king, bishop and knight versus a lone king) endgame. It is important to note that at the beginning of the process the expert was unable to express such precise instructions on his[1] own and was even unaware of some of the important concepts that were later used in the instructions.

The paper is organized as follows. We first present the obtained textbook instructions for teaching the KBNK endgame. Section 3 explains the guidelines for interaction between the machine and the expert in order to obtain a human-understandable rule-based model for playing a chess endgame, and how the instructions were derived semi-automatically from our rule-based model for KBNK. Note, however, that the details of the algorithm for obtaining the model are not a subject of this paper. In Section 4, we present the evaluation of the instructions by several renown chess teachers, and an evaluation of human-like style of play generated by our method by four international grandmasters. Finally, we give some conclusions and intentions for further work. The rule-based model for KBNK, the description of the algorithm and example games containing automatically generated instructions can be found in a web appendix at `http://www.ailab.si/matej/KBNK/`.

[1] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

2 Semi-automatically Derived Instructions for the Bishop and Knight Checkmate

We present here the instructions in the form of goals for delivering checkmate from any given KBNK position. These instructions were semi-automatically derived from the tablebases. In the following hierarchical set of goals, to deliver a checkmate successfully , the chess-player is instructed always to try to execute the highest *achievable* goal listed below. The goals are listed in order of preference, goal 1 being the most preferred. The chess-player is expected to know how to avoid stalemate, piece blunders, and threefold repetitions. Apart from descriptions of the eleven goals we also illustrate most of the concepts behind them.

Goal 1: Deliver Checkmate. A checkmating procedure is the following: two consecutive checks with the minor pieces are delivered, the later one resulting in one of the two types of checkmate positions shown in Fig. 1.

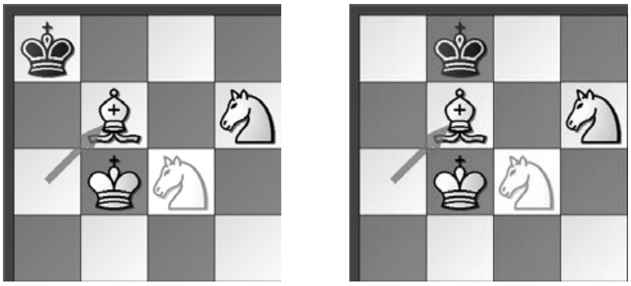


Fig. 1. Checkmate can be delivered by the bishop or the knight, always in the corner of the bishop's color ("right" corner). Each arrow indicates last bishop's move.

Goal 2: Prepare the Knight for Checkmate. This goal applies when the king and the bishop restrain the defender's king to only two squares: the corner square and a square on the edge of the board right beside the corner square (see Fig. 2). The task of the attacker is to prepare the knight so that it is ready for the checkmating procedure.

Goal 3: Restrain Defending King to a Minimal Area Beside The Right Corner. The task of the attacker is to take squares away from the defending king until it is driven to the edge of the board, and consequently to the corner square. The chess-player is advised to aim for the type of position shown in Fig. 2 where the king and the bishop restrain the defending king to a minimal area beside the right corner.

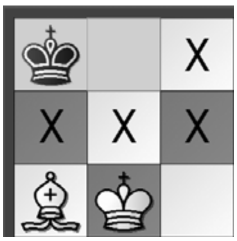


Fig. 2. A minimal area

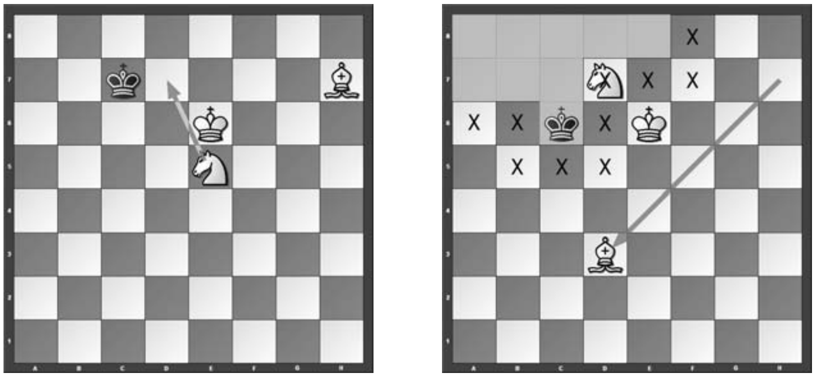


Fig. 3. In the position shown in the left diagram, the attacking side could build the barrier in the following manner: 1.Ne5-d7 Kc7-c6 2.Bh7-d3!, leading to the position on the right. The area around the right-colored corner to which the defending king is confined, could be squeezed further, *e.g.*, after 2...Kc6-c7 3.Bd3-b5.

Goal 4: Build a Barrier and Squeeze Defending King. The attacker is advised to build *a barrier* that holds the defending king in an area beside the right-colored corner. When such barrier is built, the attacker should aim to squeeze the constrained area in order to restrain the defending king (see Fig. 3).

Goal 5: Approach Defending King from Central Side. A part of the basic strategy is to drive the opposing king to the edge of the board. In order to achieve this, it is beneficial for the attacking side to occupy squares closer to the center of the chessboard then the defending king does. The attacker should aim to approach the opposing king from the central side of the board.

Goal 6: Block the Way to the Wrong Corner. When the defender's king is already pushed to the edge of the board, the attacker's task is to constrain as much as possible the defending king's way to the wrong-colored corner. At the same time, the attacker should try to keep restraining the king to the edge of the board. Fig. 4 shows an example of a typical position that often occurred in simulated games. It is called the wrong-corner position.

Goal 7: Push Defending King towards the Right Corner. The attacker is advised to push the defending king towards the right-colored corner, at the same time not allowing it to move further away from the edge of the board (see Fig. 5).

Goal 8: Push Defending King towards the Edge. The attacker is advised to arrange the pieces in such a way so that the defending king is pushed towards the edge of the board, and cannot immediately increase the distance from the edge.

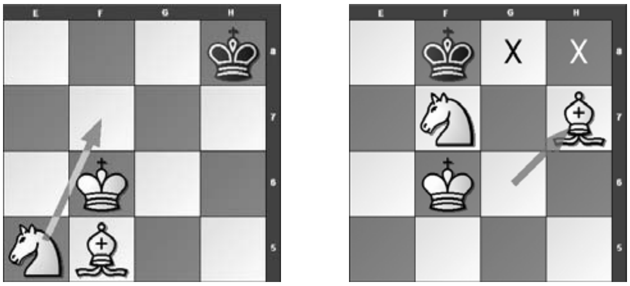


Fig. 4. In the position on the left, white pieces lure the defending king out of the wrong corner: 1.Ne5-f7+ Kh8-g8 2.Bf5-g6 Kf8 (note that this is the only available square, since h8 is attacked by the knight) 3.Bh7! The last move in this sequence takes under control square g8, and sets up the blockade one square farther from the wrong corner.

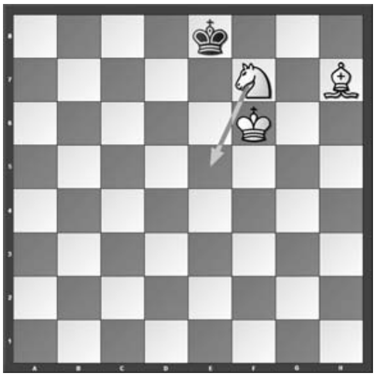


Fig. 5. Black king's distance to the desired corner (a8) should decrease, and the black King should not be allowed to move away from the edge of the chessboard. This is achieved by the move 1.Nf7-e5, and black cannot resist white's goals: even after sub-optimal 1...Ke8-f8 (optimal move according to tablebases is 1...Ke8-d8) white could play 2.Ne5-d7+ Kf8-e8 3.Kf6-e6 and black should move closer to the right corner with the only available move 3...Ke8-d8. After 4.Ke6-d6 Kd8-e8 5.Bh7-g6† the wrong corner position is repeated; it is shifted over two files.

Goal 9: Bring the King Closer to the Defending King. The attacker is advised to move the king closer to the opposing king.

Goal 10: Bring the Knight Closer to the Defending King. The attacker is advised to bring the knight closer to the defending king.

Default Goal: Keep the Kings Close. If none of the above goals is achievable, at least keep the king as close as possible to the defending king, and - if possible - strive for the opposition of the kings.

2.1 Example Games with Automatically Generated Suggestions

The instructions given in the previous subsection are accompanied by example games containing automatically generated instructions. An instruction is given each time the previous suggested goal was accomplished. These games serve to illustrate how the teaching process would run with the help of a computer. The student would first read the instructions and then be presented a random position to play against the computer. At any point in the game, the computer is able to give an appropriate suggestion to the student in the form of a goal to accomplish. These suggestions/goals could be further augmented by occasionally displaying a side diagram containing the position associated with the given goal. We give the games in a web appendix at `http://www.ailab.si/matej/KBNK/`.

3 The Process of Synthesizing Instructions

Below we have partitioned the description of our process of synthesizing the instructions into six sections.

3.1 Basic Description of Our Approach

As already mentioned, the rules were induced by a recently developed method for goal-based rule induction [9]. This method extracts a strategy for solving problems that require search (like chess, checkers etc.). A strategy is an ordered list of goals that lead to the solution of the problem, similar to an advice list in Advice Languages [10]. These goals can then be used to teach a human, who is incapable of extensive search, how to act in these domains and be able to solve these problems by following suggested goals. The method combines ideas from the Argument Based Machine Learning (ABML) [11] with specialized minimax search to extract a strategy for solving problems that require search.

3.2 Obtaining Knowledge from Domain Expert

In order to obtain meaningful and human-understandable instructions, the knowledge has to be elicited from a chess expert (in our case this was a FIDE master). Each chess position is described by a set of features that correspond to some well-known chess concepts. The features are obtained by a domain expert as a result of the knowledge elicitation process.

The knowledge elicitation process is similar as in [7,8]: the domain expert and the machine learning algorithm improve the model iteratively. A typical interaction between the method and the expert is shown in Fig. 6. As a result of this particular interaction, a new attribute *king_constrained* was introduced.

3.3 Strategic Goal-Based Rules

Our hierarchical model consists of an ordered set of rules of the following form:

`IF` *preconditions* `THEN` *goal*

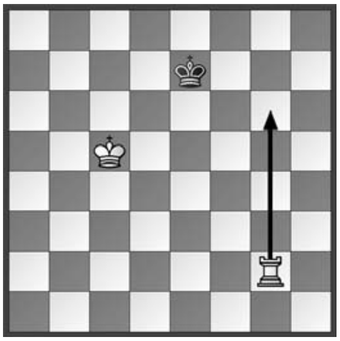


Fig. 6. Computer: *"What goal would you suggest for white in this position? What are the reasons for this goal to apply in this position?"* The expert used his domain knowledge to produce the following answer: "Black king is close to the edge of the board, but the king is not constrained by white pieces. Therefore I would suggest White to constrain black king."

The rule's *preconditions* and *goals* are both expressed by using the aforementioned features. The method used the expert's argument given in Fig. 6 to induce the following rule:

```
  IF $edist < 3$ AND $king\_constrained = false$
THEN $king\_constrained = true$ AND $edist$ should not increase
```

where *edist* is the distance between black king and the edge of the board. The subgoal *edist should not increase* was added by the computer. The method recognized that allowing Black to move away from the edge of the board would increase the distance to mate. The expert can accept or reject such suggestions before the rule's acceptance, but doing so it is important to rely on his *common knowledge* about the domain.

Preconditions can be a conjunction of various conditions (if none are given, the goal is tried each time when no higher goal in the hierarchy is achievable). Similarly, a goal is a conjunction of subgoals, where a subgoal can specify the desired value of an attribute (*true*/*false*, $<$, $>$, etc.), its optimization (*minimize*, *maximize*), and any of four possible qualitative changes: *decrease*, *increase*, *not decrease*, *not increase*. Each rule should contain exactly one progressive subgoal (as is the change of the value of *king_constrained* in the above rule). Note that a subgoal *edist should not increase* is not progressive, since it allows a chess-player to maintain merely the status, not progressing towards delivering checkmate.

3.4 Allowing Non-optimal Play

Often, *counter examples* are detected by the method, and presented to the expert. Counter examples are positions where the goal can be achieved, but the resulting play nevertheless leads to *increased* distance to mate. Among such

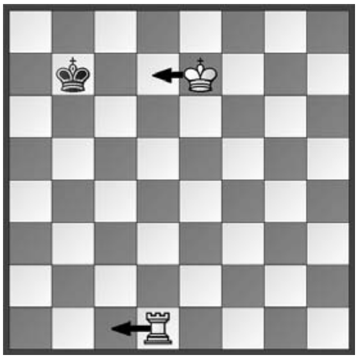


Fig. 7. Computer: *"Would you admonish a student if he or she played 1.Rd1-c1 in this position?"* The expert found this move to be perfectly acceptable. Despite of its non-optimality: from the tablebase point of view 1.Ke7-d7 is a much better move - 1.Rd1-c1 (the worst possible execution of the suggested goal) achieves mate in 11 moves whereas after 1.Ke7-d7 only 6 moves are necessary (after 1...Kb7-b6 2.Rd1-d5!).

positions, the one with the highest distance to mate is chosen as the key counter example. Figure 7 illustrates.

Since human players typically choose a longer path to win by systematically achieving intermediate goals, the expert is instructed to accept counter examples where such subgoals are executed, although they often do not lead to optimal play in the sense of shortest win against best defence. However, the resulting play in counter examples should lead to overall progress towards achieving the final goal of delivering checkmate. Constraining the black king in the above counter example was judged to lead to such progress.

The expert may also find the execution of the goal in a counter example to be unacceptable. In this case, he may add, modify, and/or remove any of the preconditions and subgoals. Again, doing any of these, it is important that the expert relies on his common knowledge about the domain.

3.5 Hierarchy of Goals

When a rule triggers, all the goals higher in the hierarchy are also taken into account. The goal is achievable when at least one of these goals can be executed regardless of the defender's play (optimal or non-optimal). Such hierarchy of goals is typical of a human way of thinking. For example, when the goal is to push the defender's king towards the right-colored corner in the KBNK endgame and the defender resists the goal by allowing the opponent to deliver checkmate (that would not be achievable without the opponent's help), one is expected to see such a possibility. It would be redundant to express goals in the following way: "Push the defending king towards the right corner *or deliver a checkmate, if the opponent plays badly and allows it.*"

3.6 Constructing Human-Friendly Instructions from Semi-automatically Generated Rules

The role of preconditions and non-progressive subgoals is merely to allow a computer program to detect positions where no specific rule triggers and to execute goals appropriately. All the goals in the instructions are obtained by stating only the progressive subgoal. The exception is the last, default goal, since it is desirable always to be able to give advice to the student. Let us demonstrate this on the following rule (the descriptions of the attributes are available in a web appendix at `http://www.ailab.si/matej/KBNK/`):

$$\texttt{IF}\ \textit{edist} < 1\ \texttt{THEN}\ \textit{edist should not increase}\ \texttt{AND}\ \textit{knight\_on\_edge} = \textit{false}$$
$$\texttt{AND}\ \textit{wrong\_corner\_way should decrease}\ \texttt{AND}\ \textit{wrong\_corner\_way minimise}$$
$$\texttt{AND}\ \textit{white\_king\_more\_central} = \textit{true}$$

The precondition *edist* < 1 enables the program to try to achieve the goal only when the defending king is confined to the edge of the board. The progressive subgoal is *wrong_corner_way should decrease*, so when this rule triggers (*i.e.*, this goal is achievable and no higher rule triggers) the student is given the following advice: “Block the way to the wrong corner.” It is expected from a human to recognize (at least eventually) that moving the knight to the edge, allowing the opponent to move away from the edge, and putting the king closer to the edge than the opponent’s king does not lead to progress. For a computer, such constraints are necessary to enable a sensible execution of the goals.

It is also desirable to obtain sensible diagrams and variations that are supposed to provide the most useful representation of the goals and concepts in a given domain. We obtained these by executing simulations of delivering checkmate from randomly chosen initial positions using the hierarchy of goals. The execution of goals in these simulations was optimal in the sense of minimizing the distance to mate (quickest play). For five goals, the position that occurred most frequently in the simulations, was chosen to be presented by a diagram. When several positions occurred equally frequently, two diagrams were used. Due to space limitations, we only presented the diagrams that occurred most frequently.

4 Discussion and Evaluation

The bishop and knight checkmate (KBNK) is regarded as the most difficult one of the elementary mates. Several chess books give the general strategy for playing this endgame as follows. Since checkmate can only be forced in the corner of the same color as the squares on which the bishop moves, an opponent will try to stay first in the center of the board, and then retreat in the wrong-colored corner. The checkmating process can be divided into three phases: (1) driving the opposing king to the edge of the board, (2) forcing the king to the appropriate corner, and (3) delivering a checkmate. However, only knowing this basic strategy hardly suffices for anyone to checkmate the opponent effectively.[2] Another strategy is known as *Delétang's triangles*, involving confining the lone king in a series of three shrinking isosceles right-angled triangles (pioneered by Delétang in 1923 [12]). This strategy usually takes five to ten moves longer to deliver checkmate. Since state-of-the-art endgame manuals (*e.g.*, [13]) prefer teaching the aforementioned three-phase checkmating process, we decided to aim for obtaining the rules for executing that (quicker) strategy.

To the best of our knowledge, no formalized models for KBNK endgame suitable for teaching purposes were derived by any machine-learning programs. As H.J. van den Herik et al. in 2002 (and still valid today) nicely put it: "The current state of the art of machine-learning programs is that many ad hoc recipes are produced. Moreover, they are hardly intelligible to human experts. In fact, the database itself is a long list of ad hoc recipes. Hence, the research question is how to combine them into tractable clusters of analogue positions and then to formulate a human-understandable rule." [6]

Based on the aforementioned Delétang's triangles method, van den Herik constructed a formalized model for playing KBNK endgame and successfully implemented it in a chess-playing program [14]. The knowledge in the model was partitioned into 28 patterned equivalence classes (introduced by Bramer [15]) aimed to correspond to some significant recognizable *features* of the endgame as perceived by chess-players such as "confinement in the wrong corner" and "intermediate class between the large and the middle bishop triangle" (the obtained equivalence classes and patterns are fully described in [16]). The model was derived from chess theory books, discussions with (grand)masters, and the author's experience, but without any machine-learning programs or chess-tablebases support. Similarly as with our approach, the resulting knowledge is intended to be used jointly with tree search with a maximum search depth decided in advance and does not necessarily produce optimal play. There are several important differences between [14] and our approach, the most important two of them are given below.

- Obtaining the formalized model for KBNK in [14] required *existence* of some method for playing the endgame in question (in this case, the method discovered by Delétang), while our ABML-based knowledge elicitation process provides a potential for obtaining goal-based instructions for *any* chess endgame where tablebases are available. Note that no known method for delivering checkmate exist for more complex endgames (such as KBBKN).
- Using a pattern-classes-based model leads to instructions in form of descriptions of *states* the chess-player should aim to achieve from a given position, while our strategic goal-based rules suggest the relative change (improvement) in a position given in terms of *one progressive goal* per instruction. For a student, it seems easier to memorize instructions containing a single progressive goal than a sequence of several states.

[2] For example, grandmaster Epishin (Kempinski-Epishin, Bundesliga 2001) failed to force the defending king to the appropriate corner and the game ended in a draw.

In a different attempt to obtain a formalized model for the KBNK endgame, van den Herik and Herschberg [17] strived for optimal play, using tablebases. After their very limited successes, the task of translating perfect information into a set of rules to be followed by a human being or a computer was reported to be too difficult for the state-of-the-art techniques of that time (1986).

The extracted strategy as described in Section 2 was presented to three chess teachers (among them a selector of Slovenian women's squad and a selector of Slovenian youth squad) to evaluate its appropriateness for teaching chess-players. They all agreed on the usefulness of the presented concepts and found the derived strategy suitable for educational purposes. Among the reasons to support this assessment was that the instructions "clearly demonstrate the intermediate subgoals of delivering checkmate."

We also evaluated the rules by using them as a heuristic function for 6-ply minimax search to play 100 randomly chosen KBNK positions (each requiring at least 28 moves to mate providing optimal play[3]) against perfect defender. We tested two different strategies for cases when the heuristic suggests several moves achieving the goal: either (a) a move that minimizes distance to mate (quickest play), or (b) a move that maximizes distance to mate (slowest play) was chosen. Our rules were able to achieve mate in 100% of the cases using both quickest play (average game length was 32 moves) and slowest play (average game length was 38 moves). Therefore, even with the slowest possible realization of strategic goals, the strategic rules are expected to guide a student reliably towards the final goal of achieving checkmate within the allowed 50 moves. It is worth noting that state-of-the-art chess engines such as Rybka 2.1, Zappa 1.1, and Toga II 1.3.1, when limited to a 6-ply search only, were not able to deliver checkmate within 50 moves against an optimal defender from any of the 100 starting KBNK positions.

Our ABML-based approach leads to obtaining domain's strategic goal-based rules using the same arguments and the same domain language attributes as the expert does. We therefore expect the resulting models to produce "human-like" style of play, in the sense that it would be clearly understandable by human players. As typical for humans, such play would not aspire to minimize the distance to win. To verify this hypothesis, we applied our approach to constructing strategic rules for the KRK chess endgame, where it is commonly accepted that a traditional way of delivering mate differentiates from optimal (tablebase) play.

We verified our hypothesis with a kind of Turing test. Four strong grandmasters were asked to observe 30 games: 10 games played by our KRK chess program guided only by the rules obtained with our ABML-based method, 10 games by a perfect (tablebase) player, and 10 additional games (to further complicate the evaluators' job), all facing a perfect (tablebase) opponent. They were only told that at least in some of the games the white player was a computer program, while black always defended optimally. The grandmasters were asked to express their assessment for each game to what degree (marks 1 to 10) they find the play

[3] KBNK is a 33-move game in the maximin sense, as it was established after the complete tablebases were computed by Dekker and van den Herik in 1982 [16].

to be human-like. The mark of 1 means that the attacker's play seems totally computer-like, and mark 10 means that it seems totally human-like. The average scores given to our KRK rules by the four grandmasters were 4.1, 7.1, 8.2, and 7.3, while the average scores given to tablebase player were 2.2, 3.1, 1.8, and 2.0, an obvious difference.

5 Conclusions

We developed a procedure for semi-automatic synthesis of textbook instructions for teaching the KBNK endgame, accompanied by example games containing generated instructions. The presentation of the derived strategy includes a number of concepts and key positions from this endgame that help the human learner to understand the main principles of the strategy. The key positions were detected automatically from simulated games played by the derived strategy. The key positions in Figures 4, 5, and 3 belong to a frequent sequence of seven moves in tablebase play. In our case, this sequence was reduced to the three key positions and conceptualized in terms of goals along the sequence. In contrast to memorizing the optimal sequence itself, the extracted generalization also enables correct play against sub-optimal defence. The derived strategy is human-friendly in the sense of being easy to memorize, but produces suboptimal play. In the opinion of chess coaches who commented on the derived strategy, the tutorial presentation of this strategy is appropriate for teaching chess students to play this ending.

We view the positive assessment of derived textbook instructions by chess coaches as a confirmation that our approach is able to facilitate knowledge extraction from the tablebases. We explained (1) the guidelines for interaction between the machine and the expert in order to obtain a human-understandable rule-based model for playing a chess endgame and (2) how the instructions, including illustrative diagrams, could be derived semi-automatically from such a model.

Our next goal is to create a computer tool for teaching the KBNK endgame. All the main ingredients are already available as described in this paper, and all that remains is to package them into an actual application as described in Sect. 2.1. A second goal is to mimic this approach for a much harder endgame, namely KBBKN.

References

1. Thompson, K.: Retrograde analysis of certain endgames. International Computer Chess Association Journal 9(3), 131–139 (1986)
2. Roycroft, A.J.: Expert against oracle. In: Hayes, J.E., Michie, D., Richards, J. (eds.) Machine Intelligence, vol. 11, pp. 347–373. Oxford University Press, Oxford (1988)
3. Nunn, J.: Secrets of Minor-Piece Endings. Batsford (1995)
4. Nunn, J.: Secrets of Pawnless Endings. Gambit Publications Limited (2002)

5. Fürnkranz, J.: Machine learning in games: A survey. In: Fürnkranz, J., Kubat, M. (eds.) Machines that Learn to Play Games. Nova Scientific Publishers, New York (2001)
6. van den Herik, H., Uiterwijk, J., van Rijswijck, J.: Games solved: Now and in the future. Artificial Intelligence 134, 277–311 (2002)
7. Možina, M., Guid, M., Krivec, J., Sadikov, A., Bratko, I.: Fighting knowledge acquisition bottleneck with argument based machine learning. In: The 18th European Conference on Artificial Intelligence (ECAI), pp. 234–238 (2008)
8. Guid, M., Možina, M., Krivec, J., Sadikov, A., Bratko, I.: Learning positional features for annotating chess games: A case study. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 192–204. Springer, Heidelberg (2008)
9. Možina, M., Guid, M., Sadikov, A., Bratko, I.: Goal-Based Rule Learning. Technical report, Faculty of Computer and Information Science, University of Ljubljana, Slovenia (2009), http://www.ailab.si/matej/KBNK/GBRL.pdf
10. Bratko, I.: Knowledge-based problem-solving in AL3. Machine Intelligence 10, 73–100 (1982)
11. Možina, M., Žabkar, J., Bratko, I.: Argument based machine learning. Artificial Intelligence 171(10/15), 922–937 (2007)
12. Delétang, D.: Mat avec le fou et le cavalier. Las Stratgie 56(2), 25–32 (1923)
13. Dvoretsky, M.: Dvoretsky's Endgame Manual, 2nd edn. Russell Enterprises, Inc. (2008)
14. van den Herik, H.J.: Representation of experts' knowledge in a subdomain of chess intelligence. In: IJCAI, pp. 252–255 (1983)
15. Bramer, M.: Representation of knowledge for chess endgames: towards a self-improving system. PhD thesis, The Open University: Faculty of Mathematics, Milton Keynes, England (1977)
16. van den Herik, H.J.: Computerschaak, Schaakwereld en Kunstmatige Intelligentie. PhD thesis, Delft University of Technology, Academic Service, The Hague (1983)
17. van den Herik, H.J., Herschberg, I.: Omniscience, the rulegiver? In: Pernici, B., Somalvico, M. (eds.) Proceedings of L'Intelligenza Artificiale Ed Il Gioco Degli Scacchi, III Convegno Internazionale, pp. 1–17 (1986)

Incongruity-Based Adaptive Game Balancing

Giel van Lankveld, Pieter Spronck, H. Jaap van den Herik,
and Matthias Rauterberg

Tilburg centre for Creative Computing
Tilburg University, The Netherlands
{g.lankveld,p.spronck,h.j.vdnherik}@uvt.nl, g.w.m.rauterberg@tue.nl

Abstract. Commercial games possess various methods of game balancing. Each of them modifies the game's entertainment value for players of different skill levels. This paper deals with one of them, viz. a way of automatically adapting a game's balance which is based on the theory of incongruity. We tested our approach on a group of subjects, who played a game with three difficulty settings. The idea is to maintain a specific difference in incongruity automatically. We tested our idea extensively and may report that the results coincide with the theory of incongruity as far as positive incongruity is concerned. The main conclusion is that, owing to our automatically maintained balanced difficulty setting, we can avoid that a game becomes *boring* or *frustrating*.

1 Introduction

The main goal of many commercial computer games is to provide entertainment to the player. To support player-experienced entertainment, the gaming industry has invested substantial efforts in improving game attributes that contribute to this goal. In particular, it is remarked that attributes such as graphics, animation, and physics have seen a rapid increase in technical detail and accuracy over the past two decades. Although the attributes mentioned have an actual impact on the entertainment value of a game, they are still a relatively small factor in this respect. More important for the entertainment value are elements such as design, diversity, story, game balancing, and gameplay.

In this paper we deal with game balancing, which is defined as the adaptation of the game difficulty towards a player's skill. More specifically, we focus on the relationship between a game's complexity and a player's abilities. In a well-balanced game, a player is challenged by the complexity of the game, but not to the extent that he[1] becomes frustrated. It is generally assumed that a balanced game has a higher entertainment value than one that is too easy or too hard [3]. If a game is considered to be balanced, then for the purpose of entertainment that game should aim at remaining balanced from start to end, even while the player learns to do better at handling specific game challenges. However all players are

[1] For the sake of brevity, we use 'he' and 'his' whenever 'he or she' and 'his and her' are meant.

different in their skills and learning abilities, therefore meeting this aim means adapting the game's complexity automatically to the observed player skill.

The incongruity theory [7] states that every context (such as a game) has a level of complexity. To deal adequately with a context humans have internal models of the varying levels of complexity. Incongruity is defined as is the difference between the complexity of a context and the complexity of the internal human model of the context. When the incongruity is too large, the human loses interest in the context, i.e., he loses interest in the game. This means that the entertainment value of the game decreases.

In this paper we apply the incongruity theory to game balancing. To do so, we measure the incongruity while a game is being played, and adapt the challenge of the game automatically to maintain the incongruity at a constant level. This level should be one that the human player experiences as entertaining, regardless of skills and capabilities. For our investigations we present a new game called *Glove*, developed in our laboratory. *Glove* contains a novel approach to keep incongruity at a desired level. According to the incongruity theory, a large incongruity is less entertaining than a small incongruity.

The outline of the paper is as follows. In Section 2 we provide background information on game balancing and incongruity theory. Section 3 describes *Glove*, including our approach to tactical balancing. The experimental setup is given in Section 4. Our results are presented in Section 5 and discussed in Section 6. Finally, in Section 7 we provide our conclusions and look at future work.

2 Background

In this section, we discuss the existing work currently done on game balancing (Subsection 2.1), and provide details on the incongruity theory (Subsection 2.2).

2.1 Game Balancing

Commercial games usually provide a manual way of setting the difficulty at the start of a new game. This method sometimes results in an inadequate difficulty setting, e.g., if the player makes an unsuitable choice or if his skills improve during play [12]. For example, the commercial game *Max Payne* features what game developers refer to as dynamic difficulty adjustment (DDA). The DDA monitors the amount of damage received, and adjusts the player's auto-aim assistance and the strength of the enemies [13]. DDA is easily recognised by the players, which implies that it breaks the flow of the game [2]. Therefore, DDA may cause players to accept extra damage to prohibit an increase of the difficulty.

Recently, computer science researchers have started to investigate methods that measure the entertainment value of a game [1,10,11], and adapt the game automatically in order to increase entertainment [4,8]. Yannakakis [9] describes two ways of optimising player enjoyment, namely implicit and explicit optimisation. In implicit optimisation, machine learning techniques such as reinforcement learning, genetic algorithms, probabilistic models, and dynamic scripting,

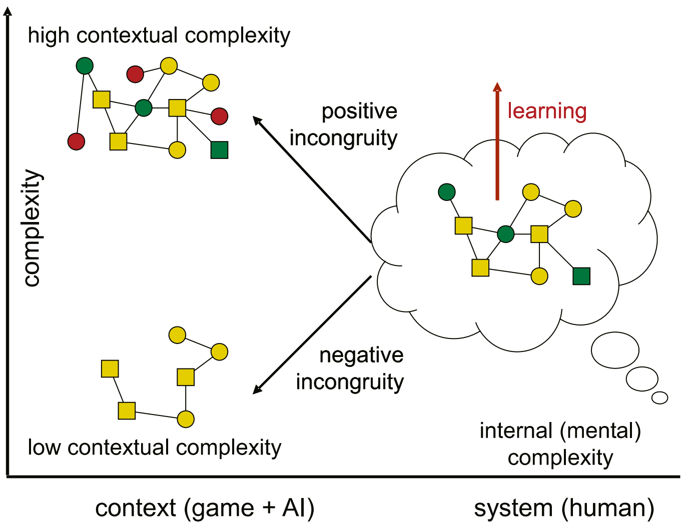


Fig. 1. Schematic representation of incongruity

are used for optimisation. He also describes user modelling techniques as explained in interactive narration. In explicit optimisation he describes adaptive learning mechanisms that optimise what he calls *user-verified ad-hoc entertainment*. Two well known techniques are: (1) neuro-evolution mechanisms and (2) player modelling through Bayesian learning.

2.2 Incongruity Theory

People continuously form mental models about the world in which they live. These models allow them to estimate how the world is going to react to different types of interaction and how the world will change over time. Incongruity theory attempts to explain the emotions that arise during interaction between these models and the world. For a proper understanding of this paper we introduce below the terminology used in the literature on incongruity.

Incongruity theory uses the term '*context complexity*' to describe the complexity of the world or a part of the world. The term '*system complexity*' is meant to describe the complexity of a mental model that a person has of the world. Complexity can be *high*, *intermediate* or *low* for both the context and the system. The term 'incongruity' is used to describe the difference in complexity between the system and the context. When system complexity is higher than context complexity we speak of negative incongruity. When context complexity is higher than system complexity we speak of positive incongruity. A schematic visualisation of these concepts can be found in figure 1.

According to the incongruity theory, the difference between the system complexity and the context complexity give rise to three types of emotion: boredom, frustration, and pleasure. *Boredom* is a feeling of reduced interest, it arises with high negative incongruity. *Frustration* is a feeling of annoyance or anger, it arises with high positive incongruity. According to Rauterberg [7], in situations of large negative incongruity, people start to look for new a stimulation. *Pleasure* is a feeling of entertainment, it arises when context complexity is roughly equal to or slightly higher than system complexity.

Learning is stimulated in situations where incongruity is positive: it raises the system complexity. Thus, learning can bring players from a large positive incongruity via low or no incongruity to negative incongruity.

In a game context we can easily identify the concepts of incongruity. First the difficulty of the game is equivalent to the context complexity. Then, the player's skill at the game is equal to the system complexity. If the player's skill is too low in order to cope with the current game difficulty setting, we may speak of negative incongruity, which is expected to result in frustration. If the player's skill is too high in comparison to the game difficulty, we may speak of positive incongruity, which is expected to result in boredom. When the player's skill matches the game difficulty, there is low or no incongruity and a feeling of pleasure or, otherwise stated, entertainment is expected to occur.

Because system complexity is part of the human mind there is usually no direct way to measure system complexity in games. One possibility is to measure the complexity of the player's behaviour in a game and infer the system complexity from that. Rauterberg [7] states that low system complexity will lead to the player's behaviour being largely determined by heuristics, while expert players, with high system complexity, use other, more straightforward methods.

3 Glove

In our experiments we use a new game called *Glove*. It is derived from the classic game *Gauntlet*. In this section we describe the *Glove* game world (Subsection 3.1), and the balancing mechanism built into the game (Subsection 3.2).

3.1 Game World

Glove (depicted in Figure 2) is a two-player turn-based game between a (human) player and a computer system. The player controls a knight. The knight is placed in a world that consists of 2000 cells. The world is 10 cells high, and 200 cells wide. Each cell is either passable (grass), or impassable (water or mountain). The knight occupies one cell. The world also contains enemies, each of which also occupies one cell only.

The knight starts at the leftmost end of the world, and his goal is to reach the rightmost end of the world. The game ends in victory for the player when the knight reaches the goal. It ends in a defeat for the player when the knight dies before reaching the goal (i.e., the computer system wins). A knight dies when he has no health left. Health is measured in hitpoints, of which the knight has 100 at the start of the game. As soon as the number of hitpoints reaches zero, the knight dies. On each turn, the player can let the knight perform one of two actions: he can either *move*, or *attack*.

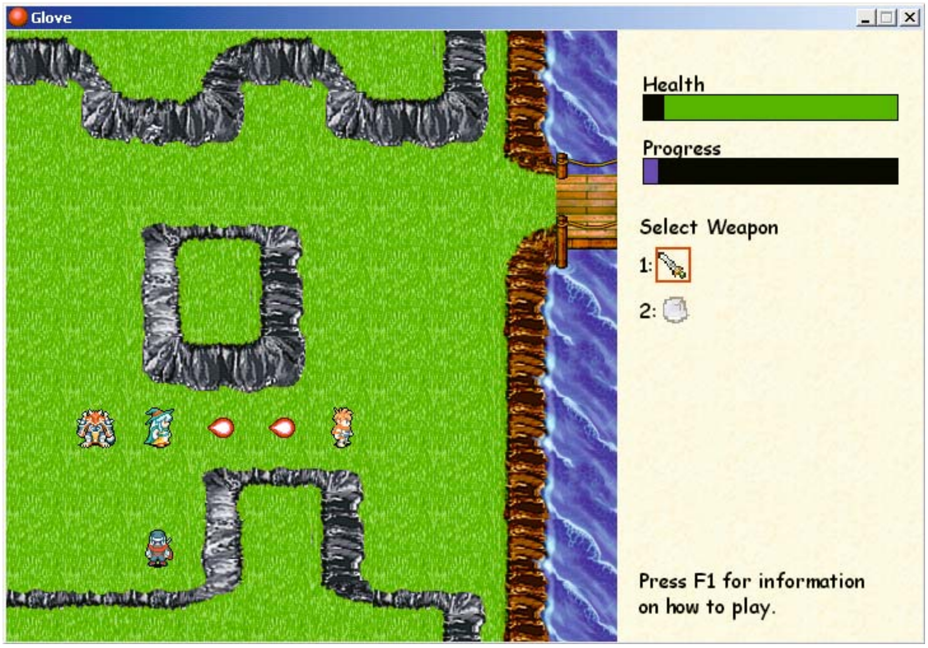


Fig. 2. Glove

When moving, the knight leaves the cell that he occupied, and moves over to any of the eight adjacent cells. Each move costs the knight 0.5 hitpoint. This means that if he moves steadily and unobstructed through the world, he has exactly enough health to win the game.

When attacking, the knight executes an attack to one of the eight adjacent cells. He can either attack with his sword that he always carries with him, or with a rock, which he may have picked up in the game world (by moving over it). The knight can carry at most one rock at a time. The difference between attacking with the sword and a rock, is that the rock actually attacks two cells, namely the cell which is attacked, and the one directly behind it (according to the movement of the knight). There are three types of enemies that can attack the cell containing the knight (dragon, ninja, and witch, see below). If an attacked cell contains an enemy, the enemy dies. Upon dying, the enemy leaves behind a health token, which the knight may pick up (by moving over it). This grants the knight 5 hitpoints (they are added to the knight's current amount up to a maximum of 100).

There are three different kinds of enemies in the world, a number of which are spawned at regular intervals. Each time that an enemy attacks and hits the knight, the knight becomes damaged and loses 5 hitpoints. The three types of enemies differ in their behaviour and their abilities. The three enemy types are the following.

1. **Dragon:** The dragon approaches the knight using a shortest-path method. The dragon is visible. When the dragon is next to the knight, he attacks the cell in which the knight resides. Arguably, the dragon's behaviour is the easiest behaviour of all three to deal with by the player.
2. **Ninja:** The ninja has the same basic behaviour as the dragon, but he has an additional ability: he can become invisible. So, he will use this ability when he is within a certain range of the knight, and will remain invisible for a certain number of turns. As soon as he attacks the knight, he will become visible again. The ninja's behaviour is reasonably predictable, even when invisible, for players who possess a good mental model of the game.
3. **Witch:** The witch approaches the knight in the same way as the other two enemy types, but stops when she is within a distance of three cells of the knight. At that point, she will start to throw one fireball per turn in the direction of the knight. Fireballs move at a speed of one cell per turn. When there are few enemies on the screen, fireballs can usually be avoided easily. However, the knight must approach the witch to be able to attack her, at which time avoidance may be difficult. Damage statistics produced in the experiment lead us to conclude that the witch is the hardest enemy to deal with.

3.2 Balancing Glove

Interaction with the game world is limited to move and attack actions, and there is little diversity in the challenges that the player faces. This is done on purpose. The aim of *Glove* is to provide the player with an entertaining experience, by *only* varying the number and types of enemies with which the knight is confronted.

The basic game has three difficulty levels, named *easy*, *balanced*, and *hard*. While it is possible to add more difficulty levels, for the present experiment these three were deemed sufficient. When the difficulty is easy, the game aims at having the knight win the game with about 50% of his health remaining. When the difficulty is hard, the game aims at having the knight lose the game when he has progressed through about 50% of the game world. When difficulty is balanced, the game aims at having the knight experience a narrow victory or a narrow defeat. The game accomplishes the result envisaged by controlling the number and types of spawned enemies. In this way the easy and hard levels try to keep the incongruity stable and high, while the balanced level tries to keep incongruity stable and at a minimum.

For each enemy type, the game retains the *average damage* in hitpoints that the enemy type involved has in relation to the knight. The number of hitpoints can be positive or negative (or zero). If the number is positive, it means that the knight on average loses health due to an encounter with this enemy type. If the number is negative, it means that the knight on average gains health due to an encounter with this enemy type. Gaining health is possible because the enemies leave behind health tokens upon dying, and it is certainly possible to kill an enemy without the enemy being able to damage the knight.

The net result of the spawning procedure is that between 2 and N enemies are spawned, N being a number that depends on the difficulty setting and the current progress of the knight. The spawned enemy types are such that, according to past experience with the enemies, the knight is expected to lose or gain the amount of health needed to achieve the goal of the difficulty setting, regardless of the player's skills.

The player can only see a part of the game world. He can see a 10 by 10 area centered around the knight. Enemies are spawned just outside the knight's vision, every 10 cells that the knight has progressed towards the right end of the world. The number and types of spawned enemies are determined by the game based on (1) the difficulty level, (2) the knight's progress, (3) the knight's health, and (4) the average damage numbers. To spawn enemies, the following algorithm is used (in pseudo-code).

```
procedure spawnEnemies
begin
    needed_health := getNeededHealth( getCurrentProgress() ) +
        getModifier( getDifficulty() );
    health_to_lose := getCurrentHealth() - needed_health;
    expected_health_loss := 0;
    spawned := 0;

    while
        (spawned < 2) or
        ((expected_health_loss < health_to_lose) and
        (spawned < getMaxSpawn(
            getLastSpawned()+1, getDifficulty() )))
    do
    begin
        enemyType := spawnRandomEnemy( health_to_lose );
        health_to_lose := health_to_lose -
            getAverageDamage( enemyType );
        spawned := spawned + 1;
    end;
end;
```

This algorithm uses the current game state and the settings of the game to determine the amount and type of enemies to spawn. It consists of three parts. The first part considers (1) the progress that the knight has already made through the game world and (2) the health that he is expected to have. Part two deals with spawning enemies until the combined enemy difficulty is at the level required by the game settings. Finally, part three is used for smoothing the increase of difficulty. It ensures that the amount of spawned enemies increases gradually rather than instantly. What follows now is a detailed explanation of eleven subroutines used in the algorithm.

`getCurrentProgress()` returns a percentage that expresses how far the knight has progressed through the game world. `getNeededHealth()`, uses the knight's current progress as a parameter, and returns the number of hitpoints that the knight needs when traversing the remaining part of the game world, if unobstructed by enemies. `getDifficulty()` returns the difficulty level (easy, balanced, or hard), and `getModifier()`, has the difficulty level as a parameter; it returns a number that is 50 for the easy difficulty level, 5 for balanced, and −50 for hard. `getCurrentHealth()` returns the current health of the knight. The result of the first two lines of the algorithm is that `health_to_lose` contains the number of hitpoints that the knight should lose for the game to reach the goal determined by the difficulty level. This number can be negative, which indicates that the knight should actually gain health.

`spawnRandomEnemy()` spawns an enemy. This function has `health_to_lose` as a parameter, by which it determines (1) whether it should spawn enemies that are likely to gain the knight some health, or (2) whether it should spawn enemies that cause the knight to lose health. To avoid the algorithm becoming into an endless loop, when the function should make the knight gain health, it will always allow dragons to be spawned; moreover, when it should make the knight lose health, it will always allow ninjas and witches to be spawned. We note that with more enemies, it is harder to avoid damage; even if the player has reached a skill level in which he manages to gain health from all enemy types, he will consider the game harder if he will be surrounded by more of them.

`getLastSpawned()` returns the number of enemies that were spawned at the last time. `getMaxSpawn()` returns the maximum number that can be spawned. It has two parameters, the first is a maximum that cannot be exceeded, and the second is the difficulty level, which is used to determine a maximum number: 5 for easy, 7 for balanced, and 9 for hard. Finally, `getAverageDamage()` returns the average damage done by the enemy type that is used as the parameter.

4 Experimental Setup

To test the effect of our game-balancing approach, and to investigate whether *boredom* and *frustration* are indeed associated with a decreased entertainment value and with increased incongruity, we requested a number of human test subjects to play *Glove*. The experimental setup was as follows. Each human subject played the game four times. The first time was a training run, in which the player should experience the game controls. In the training run, at each spawn point the same three enemies are spawned, namely one of each type. The player was allowed to interrupt the play whenever he wanted, to start the actual experiment.

In the actual experiment, the subject had to play the game three times, viz. once with an easy difficulty setting, once with a balanced one, and once with a hard one. The order in which the difficulty settings were presented to the subject was varied, each possible order being tested about an equal number of times. The subject was not aware of the difficulty setting of his current game. A digital questionnaire was presented to the subject after each game.

The questionnaire contained a total of 26 items. The items were all in Dutch because our subjects were all Dutch native speakers. The items fell into five categories, namely *boredom*, *frustration*, *pleasure*, *concentration*, and *curiosity*. *Boredom*, *frustration* and *pleasure* are the experiences expected to occur during play according to the incongruity theory. *Concentration* and *curiosity* items were added to assess whether the subjects were fully focused on the game, rather than other thoughts or things in their surroundings. Each item was administered using a seven-point Likert scale [6]. The seven points range from "does not apply to me at all" to "completely applies to me". The English translation of the Dutch questionnaire can be found in the appendix.

In our preliminary experiments 24 subjects participated. The subjects' age ranged from 16 to 31 years. The subjects were selected from family, friends, and the student population. All were Dutch native speakers. None of them had prior knowledge of the game before playing. The subjects had a varying background, and varying experiences with computers and games. The exact subject background did not matter for this experiment, since the game balances itself automatically to the skills of the player.

5 Results

On the questionnaires, scores ranged from zero (0) to six (6) on a Likert scale. So, assuming a continuous scale then the average would be 3. For each subject and then for each category, the average of the answers to the questions belonging to the category was calculated. Subsequently, for each of the difficulty settings, the means of these averages over all test subjects were calculated. The means are presented in Figure 3.

For a statistical analysis of the results, we had to remove one subject from the pool because of an input error, leaving 23 subjects (N = 23). To compare the means of the variables, an ANOVA is sufficient. However, (1) we had multiple conditions (easy, balanced, hard) for the prediction of the five variables and (2) we applied all three test conditions to each subject, therefore a repeated-measures MANOVA test was needed. Straightforwardly, using an ANOVA test would have ignored possible interaction and repetition effects.

The repeated-measures MANOVA multivariate test produced significant effects ($P < 0.01$). Thereafter a post-hoc univariate analysis and contrast analysis were performed in order to examine (1) the differences between the five measured variables and (2) the differences of the difficulty on these variables.

We found that the effect of order was not significant ($P > 0.05$). A subsequent analysis was performed to see if there were significant effects of experience with computer games. This effect was also not significant ($P > 0.05$).

Next, we tested the effect of the difficulty setting on each of the five categories of the questionnaires. We found no significant results for the categories *boredom*, *concentration*, and *curiosity* ($P > 0.05$ for all of them). However, we *did* find significant effects for the categories *frustration* ($P < 0.01$) and *pleasure* ($P < 0.05$).

Contrasts showed that for the category *frustration*, the differences between easy and balanced, and between balanced and hard were both significant ($P < 0.01$). In particular, we found that *Glove* is significantly more frustrating for a balanced difficulty compared to an easy difficulty, and significantly more frustrating for a hard difficulty compared to a balanced difficulty. The estimated

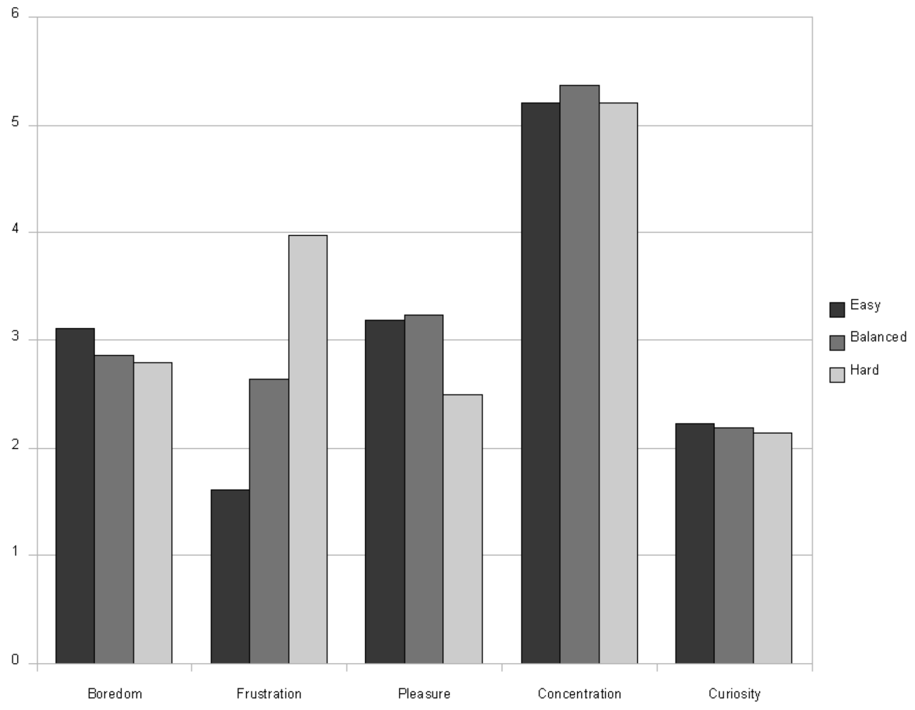


Fig. 3. Means for each category per difficulty setting

marginal means for the category *frustration* were 1.64 for easy difficulty, 2.67 for balanced difficulty, and 4.01 for hard difficulty.

For the category *pleasure* we found significant effects for the difference between balanced and hard difficulty ($P < 0.05$). In particular, we found that *Glove* provides significantly more pleasure for a balanced difficulty than for a hard difficulty. We did not find a significant effect for the difference between an easy and a balanced difficulty. The estimated marginal means for the category *pleasure* were 3.24 for easy difficulty, 3.25 for balanced difficulty, and 2.50 for hard difficulty.

Our tests show that our approach to game balancing, based on incongruity, can influence both the frustration level and the entertainment level of a game. The results reproduce the incongruity theory findings that a high positive incongruity is correlated to frustration, and that, at least for *Glove*, a balanced difficulty setting is more entertaining than a hard difficulty setting.

6 Discussion

The results of our experiments reproduce incongruity theory predictions in part rather well. The frustration effect follows the expectations of incongruity theory, while boredom (which should be significantly higher for easy difficulty) does not follow the expectations. The entertainment effect is also according to expectations, at least for the balanced and hard difficulty settings.

It is likely that entertainment would also be as expected for easy difficulty, if easy difficulty was considered to be boring by the test subjects. Therefore it is interesting to examine why the easy difficulty setting was not found to be boring. We did not actually investigate this issue, but offer two possible explanations. First, incongruity theory was originally applied to (relatively old) web interfaces [7], and the increased visual and functional interactivity of our game, even in its simplicity, might cause a sufficiently high increase in complexity to be interesting in all modes of difficulty. Second, it is definitely possible that our easy difficulty setting is still sufficiently complex to create positive incongruity. In future work, we will examine this possibility by introducing a 'very easy' difficulty setting, in which the knight is confronted with just a handful of enemies, and does not lose any health moving.

We believe that our method of adaptive game balancing overcomes some of the problems of which commercial games suffer with their method of difficulty scaling, as our balanced difficulty setting manages to avoid that the game becomes boring or frustrating.

7 Conclusions and Future Work

In this paper we examined (1) the relationship between game balancing and incongruity, and (2) how adaptive game balancing can be used to increase the entertainment value of a game. For our game *Glove*, we found that *frustration* increases with difficulty, while the *entertainment* remains roughly the same for easy and balanced difficulty, but drops for hard difficulty. So, we may conclude that our results coincide with the incongruity theory as far as positive incongruity is concerned. Furthermore, we may conclude that our approach to adaptive game balancing is suitable to maintain a game's entertainment value by keeping incongruity at a balanced value.

The pool of test subjects used for our experiments was relatively small, yet the results on which we base our conclusions are highly significant. Still, we could not discover significant results for all the categories which we examined. Significant results for the remaining categories might be obtained by a higher number of test subjects. Therefore, in future work, we will (1) continue our experiments with a bigger subject pool, (2) introduce a 'very easy' difficulty setting, to examine whether the boredom expectations of incongruity theory can also be confirmed, (3) implement our adaptive game balancing approach in an actual commercial game, and (4) test its effect on the entertainment value. Such an experiment is expected to demonstrate the applicability of our approach to commercial game developers, and may have an impact on how games are constructed in the near future.

Acknowledgements

This research was supported by the "Knowledge in Modelling" project of the Dutch National Police Force (KLPD).

References

1. Beume, N., Danielsiek, H., Eichhorn, C., Naujoks, B., Preuss, M., Stiller, K., Wessing, S.: Measuring Flow as Concept for Detecting Game Fun in the Pac-Man Game. In: Proc. 2008 Congress on Evolutionary Computation (CEC 2008) within Fifth IEEE World Congress on Computational Intelligence (WCCI 2008). IEEE, Los Alamitos (2008)
2. Csikszentmihalyi, M., Csikszentmihalyi, I.: Introduction to Part IV in Optimal Experience: Psychological Studies of Flow in Consciousness. Cambridge University Press, Cambridge (1988)
3. Charles, D., Black, M.: Dynamic Player Modelling: A Framework for Player-Centric Games. In: Mehdi, Q., Gough, N.E., Natkin, S. (eds.) Computer Games: Artificial Intelligence, Design and Education, pp. 29–35. University of Wolverhampton, Wolverhampton (2004)
4. Hunicke, R., Chapman, V.: AI for Dynamic Difficulty Adjustment in Games. In: Proceedings of the Challenges in Game AI Workshop, 19th Nineteenth National Conference on Artificial Intelligence. AAAI 2004 (2004)
5. Iida, H., Takeshita, N., Yoshimura, J.: A Metric for Entertainment of Boardgames: Its Implication for Evolution of Chess Variants. In: Nakatsu, R., Hoshino, J. (eds.) Entertainment Computing: Technologies and Applications, pp. 659–672. Kluwer Academic Publishers, Boston (2002)
6. Likert, R.: A Technique for the Measurement of Attitudes. Archives of Psychology, New York (1932)
7. Rauterberg, M.: About a framework for information and information processing of learning systems. In: Falkenberg, E., Hesse, W., Olive, A. (eds.) Information System Concepts–Towards a consolidation of views (IFIP Working Group 8.1), pp. 54–69. Chapman and Hall, London (1995)
8. Spronck, P., Sprinkhuizen-Kuyper, I., Postma, E.: Difficulty Scaling of Game AI. In: Proceedings of the 5th Internactional Conference on Intelligent Games and Simulation (GAME-ON 2004), pp. 33–37 (2004)
9. Yannakakis, G.N.: How to Model and Augment Player Satisfaction: A Review. In: Proceedings of the 1st Workshop on Child, Computer and Interaction. ICMI 2008, Chania, Crete, October 2008. ACM Press, New York (2008)
10. Yannakakis, G.N., Hallam, J.: Towards Capturing and Enhancing Entertainment in Computer Games. In: Antoniou, G., Potamias, G., Spyropoulos, C., Plexousakis, D. (eds.) SETN 2006. LNCS (LNAI), vol. 3955, pp. 432–442. Springer, Heidelberg (2006)
11. Yannakakis, G.N., Hallam, J.: Modeling and Augmenting Game Entertainment Through Challenge and Curiosity. International Journal of Artificial Intelligence Tools 16(6), 981–999 (2007)
12. http://www.casualgamedesign.com/?p=39
13. http://www.gameontology.org/index.php/Dynamic_Difficulty_Adjustment#Max_Payne

A Questionnaire (English Translation)

For the sake of clarity, the questions were translated from the original Dutch version into English.

1. In my opinion the game was user friendly
2. I am interested in how the game works
3. I had fun while playing the game
4. I want to know more about the game
5. I can easily concentrate on what I need to do during the game
6. The time passed quickly
7. I got distracted
8. I felt involved in the task
9. The game frustrated me
10. The game made me curious
11. I felt challenged
12. The time passed slowly
13. The task fascinated me
14. I was thinking about other things during play
15. I found the game to be fun
16. I was bored
17. The game was tedious
18. I would like to ask questions about the game
19. I thought the game was hard
20. I was alert during the game
21. I was day dreaming during the game
22. I want to play the game again
23. I understood what I was supposed to do in the game
24. The game was easy
25. I feel I was not doing well during the game
26. I was annoyed during play

Data Assurance in Opaque Computations

Joe Hurd[1] and Guy Haworth[2]

[1] Galois, Inc., 421 SW 6th Ave., Suite 300 Portland, OR 97204, USA
joe@galois.com
[2] School of Systems Engineering, University of Reading, UK
guy.haworth@bnc.oxon.org

Abstract. The chess endgame is increasingly being seen through the lens of, and therefore effectively defined by, a data 'model' of itself. It is vital that such models are clearly faithful to the reality they purport to represent. This paper examines that issue and systems engineering responses to it, using the chess endgame as the exemplar scenario. A structured survey has been carried out of the intrinsic challenges and complexity of creating endgame data by reviewing the past pattern of errors during work in progress, surfacing in publications and occurring after the data was generated. Specific measures are proposed to counter observed classes of error-risk, including a preliminary survey of techniques for using state-of-the-art verification tools to generate EGTs that are correct by construction. The approach may be applied generically beyond the game domain.

1 Introduction

The laws of chess in use today date back to the end of the 15th century [1], while the rules of play, which need not concern us here, are regularly revised by FIDE [2]. There have thus been 500 years of attempts to analyse the game, the last 50 years being increasingly assisted by the computer since world-class computer chess was declared to be an aim of Artificial Intelligence at the 1956 Dartmouth Conference [3].

One approach has been the complete enumeration and analysis of endgames which are small enough to be computable in practice. Heinz [4] cites Ströhlein's 1970 Ph.D. thesis [5] as the first computer construction of endgame tables (EGTs). Today, 6-man chess is essentially[1] solved [6] and the EGTs are being distributed by DVD, ftp transfer and p2p[2] collaboration [7]. With new workers, ideas, and technology already active, the prospects for 7-man chess being similarly solved by 2016 are good.

The intrinsic problem of correctness of chess endgame data is summed up in the following quotation [8].

> "The question of data integrity always arises with results which are not self-evidently correct. Nalimov runs a separate self-consistency check on each EGT after it is generated. Both his EGTs and those of Wirth yield exactly the same number of mutual zugzwangs {…} for all 2- to 5-man endgames and no errors have yet been discovered."

[1] *Essentially*, because available 6-man EGTs do not include lone Kings or castling rights.
[2] The promulgation of the Nalimov EGTs is the second most intense use of eMule.

A verification check should not only be separate but independent. While Nalimov's verification test is separate and valuable in that it has faulted some generated EGTs, it is not independent as it shares 80% of its code with the generation code [9]. Nevertheless, it should be added that Nalimov's EGTs have not been faulted to date and are widely used without question.

The problem of correctness is of course not unique to chess endgames or computer software. Famous examples include the faulty computer hardware that caused the Intel Pentium processor to divide certain numbers incorrectly [10]. In the field of mathematics, the *Classification of Simple Groups* theorem has a proof thousands of pages long which it is not feasible to check manually [11]. The recent computer-generated proof of the Four Colour Theorem [12] will bring some comfort to those who find opaque proofs by exhaustion lacking in both aesthetics and auditability.

Modern society is increasingly dependent on the integrity of its digital infrastructure, especially in a real-time, safety-critical context; globalization has led to greater homogeneity and standardised systems in all sectors, including that of Information Technology. The Internet, the Web and search engines leave their users ever more vulnerable to systemic failure, e.g., severings of vital FLAG cables [13-,14], Internet root nameserver corruption [15] and a Google search-engine bug [16].

It is therefore appropriate to look for evidence of highly assured system integrity and for tools to help to provide that manifest integrity. Chess endgames are not safety-critical but serve as a case study to demonstrate the issues of assurance management.

The main contributions here are the creation of a framework for analysing data assurance at every stage of the data life cycle, the use of this framework to analyse EGT vulnerabilities and the proposal of countermeasures and remedies. Naturally, mathematical proofs cannot apply directly to the real world, and measures have to be taken to check against the fallibilities of man and machine. However, these measures and the HOL4/BDD-based approach [17] alluded to, demonstrate and deliver the highest levels of assurance to date.

The remainder of this paper is structured as follows. Section 2 reviews sources of error in the whole process of endgame data management. Section 3 reviews the high-assurance approach taken to generating EGTs using higher-order logic. The summary, in Section 4, indicates the future prospect for generating and validating chess EGTs.

2 An Error Analysis of Endgame Data Management

It is convenient to refer here to the producer of data as the *author* and the user of that data as the *reader*. The reader may become the author of derived data by, e.g., a data-mining investigation. The lifecycle of the data, from first conception to use, is considered in three structured phases as follows.

1) Definition: the author

   – models the scenario which is to be the subject of a computation,
   – analyses the requirements and designs/implements a computation,

2) Computation:

   – the author runs the computation on a platform and generates the output,

3) Use:

   – the author manages the output: publishes, promulgates, comments,
   – the reader interprets and uses the results of the computation.

2.1 Phase 1: Definition, Design and Development

The design of the computation involves mapping the relevant characteristics of the chosen problem domain into a *representative model* on the computer. Just as a good choice of concepts and notation will facilitate a mathematical proof, the choice of language to describe the *real world* and *the model* will facilitate the faithful translation of the last, informal statement of requirements into their first formal statement. As with much of what follows, the human agent needed combines the qualities of a domain expert and a systems engineer and can carry the responsibility of ensuring that requirements are faithfully and auditably translated into systems.

Increasingly, people are working in teams – in academia, in industry, and internationally in the Open Source movement, across the web and towards the Semantic Web [18]. This further requires that the concepts in use are shared effectively and that the semantics of the language of any project are robust. In the late 1970s, a UK computer company defined a machine instruction for a new mainframe processor as 'loop while predicate is TRUE'. In the South of England, *while* means *as long as* but in the North where the processor was being manufactured, *while* commonly means *until* – the exact opposite. The second author was one of the few who realised the implications of this facet of regional language just in time.

The context in which a computer system works is important, as the system will interact with its context through the interfaces at its system boundary. Van den Herik et al. ambitiously[3] computed a KRP(a2)KbBP(a3) EGT [19], substituting complex chessic logic for unavailable subgame EGTs. This ran the risk of *model infidelity* as chess is notoriously resistant to description by a small set of simple rules [20] and they rightly caveated the results. The logic and results were indeed faulted [21], leading to refined statistics [22] which are now being compared with results generated by Bleicher's FREEZER [23] and WILHELM [24].

More subtly, a particularly efficient algorithm for computing DTM[4] EGTs [25] exploits deferred adoption of subgame data, and therefore had to be forced to compute a minimum of *maxDTM* cycles even if an interim cycle did not discover any new decisive positions.[5]

Here are three examples of model infidelity in the categories of *one out* or *boundary* errors. Wirth [26] used the software RETROENGINE which assumed that captures ending a phase of play would always be made by the winner. As a result, his depths are sometimes one ply too large. Secondly, De Koning's FEG EGT generator originally failed to note losses in 0 in endgames with maxDTM = 1 [27][6]. The so-called

[3] n.b., in 1987, computers were some 16,000 times less powerful than they are today.

[4] DTM ≡ Depth to Mate, the ultimate goal: the most common, Nalimov EGTs are to DTM. DTZ ≡ Depth to (move-count) Zeroing (move): a target metric for computing Pawnful EGTs.

[5] As for endgames such as KQKR, KRKP, KRKR, KBPK and KBBKP.

[6] FEG also suffered temporarily from the *Transparent Pawn Bug*: 'model infidelity' again.

KNNK Bug infected 7 4-man[7], 35 5-man and then many of Bourzutschky's 6-man FEG EGTs [28] before the latter spotted the problem. Thirdly, Rasmussen's data-mining of Thompson's correct EGTs [29] resulted in a small number of errors stemming from various coding slips [30-33].

A proposal here is that elements of a system should be designed to be self-identifying. There is a requirement in the next two phases, *Computation* and *Use*, that agents should confirm at the time that they are working with appropriate inputs.[8] The deepest endgames require a 2-byte cell per position in their Nalimov EGTs. However the access-code cannot determine cell-size at runtime and has to be appropriately configured for the cell-size chosen. This creates the potential for a mismatch between access-code and EGT and this inevitably occurred on occasion [35]. Most recently, Bourzutschky provided the last 16 Nalimov-format EGTs to KPPKPP by converting his FEG EGTs, and picked a 2-byte format for the two endgames, KQPK(B/R)P, [36] where Nalimov chose 1-byte.[9] Clearly, runtime checks on the actual parameters of Nalimov-style[10] EGT files would obviate several problems.

Finally, Tamplin had to manage many coding issues in porting Nalimov-originated code from one environment to another,[11] amongst which was the synchronization of parallelism, a technical issue which is becoming commonplace on multi-core platforms. It is expected but not guaranteed that this issue would be detected by independent verification testing.

2.2 Phase 2: Computation

This phase reviews hardware and software errors, and the human errors of incorrect input and inadequate verification. First, a caution against assuming the correctness of the infrastructure used for a computation. In his Turing Award lecture, Thompson [37] noted that anyone could insert their own nuances at any level of the computing infrastructure – application, collector, compiler or even commodity hardware. However, one has the reassurance that others are using the same infrastructure in a different way and there is perhaps some 'safety in numbers'.

A correct computation requires the correct input. The computation of a DTx EGT[12] can require compatible EGTs for subgames. Because of the lack of self-identification noted above, Tamplin [35, 38] had to manage his file directories with great care to ensure correctness when computing first DTZ then DTZ_{50} EGTs: the one slip was picked up by a verification run.

In the spirit of this paper, Schaeffer [39] detailed the various difficulties his team had encountered in computing 10-man EGTs. Like Tamplin, they had to manage a large set of files with great care. With regard to platform integrity, he noted not only application coding errors in scaling up from 32-bit to 64-bit working but also compiler errors in compromising 64-bit results by using 32-bit working internally. For a

[7] The seven 4-man endgames affected were KBK(B/N/P), KNK(B/N/R) and KNNK.

[8] This is just one example of the benefits of *run-time binding* [34].

[9] Another EGT conversion from 2-byte to 1-byte produced full EN/MB compatibility.

[10] e.g., game, endgame, metric, side-to-move, block/cell-size, date, version, comments …

[11] Nalimov write compilers for Microsoft, and used their non-standard features in his programs.

[12] e.g., a DTZ_k EGT computing DTZ in the context of a hypothetical k-move drawing-rule.

while, Nalimov dropped multiples of 2^{32} positions in counts of draws in his EGT statistics because of 32-bit limitations.[13,14]

Schaeffer [39] compared his checkers EGTs with those of a completely independent computation and found errors on both sides. Despite the fact that modern microchips devote a greater proportion of their real estate to self-checking, Schaeffer also noted hardware errors in CPU and RAM. He also noted errors in disks, which should give pause to think about the physics and material science of today's storage products. Checksums at disc-block level were added to prevent storage and copying errors promulgating. Nalimov EGTs were integrity-checked by the DATACOMP software, and those investing in them soon learned to check MD5SUM file-signatures as well.

With regard to software testing practice, Schaeffer [39] notes than an EGT-verification which operates only within an endgame, i.e., without regard to the positions of successor endgames, will not pick up *boundary errors* caused by misinheriting subgame information.

2.3 Phase 3: Use

The scope of the *use* phase includes errors of user cognition, data persistency and data access. Clearly, the mindsets of author and reader in, respectively, phases 1 and 3 have to be aligned if the data is not to be misinterpreted. For example, in relating to some early EGTs [40], it is necessary to remember that they are not of the now-prevalent Nalimov type, and that Thompson caveated his original KQPKQ EGT as correct only in the absence of underpromotions. Stiller [41] found an 'error' in the EGT traced to this cause. Thompson's data is for Black to move (btm) positions only, and the depth of White to move (wtm) positions is reported as the depth of the succeeding btm position in the line, i.e., not including the next wtm move and one less than what is now commonly understood to be the depth. This interface quirk nearly produced a systematic *one out* error in Nunn's 2nd edition of [42]. Further, the values reported by Thompson are either *White wins in n moves* or *White does not win*. Thus 0-1/= zugzwang positions are invisible, and 0-1/1-0 zugzwangs are not distinguishable from =/1-0 ones. As with all extant EGTs, castling is assumed not to be an option, currently reasonable as castling rights have never survived to move 49 [43], but this does mean that EGTs will not help solve some Chess Studies.[15]

There are arguments for computing EGTs to a variety of metrics [45] and therefore they need to identify their particular metric to any chess-engine using them. It can be demonstrated that when a chess-engine mistakes a DTZ EGT for a DTM EGT, it will prefer the position-depth in the current phase of play before a capture to that in the next phase, resulting in the bizarre refusal of a piece *en prise*.[16] Thus, cell-size, metric, and presumed *k*-move-rule in force must be part of the self-description of an EGT.

[13] The second author sympathises: he made a similar error in a 1970s statistics package.

[14] Nalimov also once provided an incorrect and as yet unidentified statistics file for KBPKN.

[15] e.g. r1b5/8/8/7n/8/p7/6P1/RB2K2k w Q [44]. White draws: 1. Be4 Ra4 2. Bc6 Ra6 3. g4+ Rxc6 4. gxh5 Ra6 5. h6 Bf5 6. h7 Bxf7 7. O-O-O+ K~ 8. Rd6 Ra4 (8. … Rxd6 =) 9. Rd4 etc.

[16] e.g., given 8/3Q4/8/k7/6r1/8/8/K7 w and a DTZ EGT interpreted as a DTM EGT, a chess-engine will play 1. Qf5+? Kb6 2. Qe6+? Kc5 3. Qf5+? Kc4 4. Qe6+? Kc5 (pos 3w) 5. Qc8+? Kd5 6. Qf5+? Kc4 7. Qe6+? (pos. 4b).

Disc drives built 'down to cost' are perhaps the weakest part of PCs and laptops and subject to crash: this is a strong selling point for the diskless notebook. CDs/DVDs, particularly rewritable, are prone to environmental wear and handling damage [46], and it is somewhat ironic that the ancient materials of stone, parchment and shellac are more long-lived. To check against data decay, and for data persistency, it is necessary to check that data files have not subsequently been corrupted, e.g., by file-transfer (upload, download, CD burn, or reorganisation) or even deterioration during long-term storage. File use should therefore be preceded and followed by file-signature checks on input files, and it seems surprising that this is not an inbuilt facility in computers' operating systems. RAID systems are excellent but not immune to the failure-warning system being accidentally turned off, e.g., by software update.

Incorrect file-access code can turn an uncorrupted file into a *virtually corrupted* file: the Nalimov 1-byte/2-byte syndrome is an example here. This phenomenon afflicted KINGSROW in a World Computer-Checkers Championship [47], causing it to lose an otherwise drawn game and putting it out of contention for the title.

Finally, there are some errors where the source has not been defined [39, 48]. Thompson [40] also cites errors in the KQP(g7)KQ EGT by Kommissarchik [49] but these errors[17] did not prevent this EGT from assisting Bronstein during an adjournment to a win in 1975.[18]

Following this review of the lifecycle of data, it is clear that if readers are to be assured of data integrity, authors must provide self-identifying files with file-signatures and a certificate of provenance describing the production process and the measures that have been and should be taken to ensure integrity.

3 Correct-by-Construction Endgame Tables

As the preceding section demonstrates, errors can creep in at any point in the lifecycle of data, and there is no single solution that will eliminate all errors. A pragmatic approach is to analyse the errors that have occurred, and introduce a remedy that will reduce or eliminate a common cause of errors. For example, RAIDs and file signatures are both remedies designed to tackle errors caused by faulty hard disc drives.

A common cause of errors observed in practice is the misinterpretation of EGT data. Is the context in which the reader is using the data compatible with the context in which the author computed it? Stated this way, it is clear that this problem affects a broad class of data, not just EGTs, and there is a general approach to solving the problem based on assigning meaning to the data. If data carries along with it a description of what it means, then it is possible to check that the author and reader use it in compatible contexts. A prominent example of this approach is the Semantic Web project [18], which is working towards a world in which web pages include a standardized description of their contents.

It is possible to apply this approach to EGTs by creating a standardized description of their contents, which unambiguously answers the question: what exactly does each

[17] There are btm KQQ(g8)KQ draws and wins which Komissarchik did not anticipate.

[18] Grigorian-Bronstein, Vilnius: after 60m, 8/8/8/K2q2p1/8/2Q5/6k1/8 w {=} ... 76. Qd2?? Qc6+ {77. K~ Kh1} 0-1.

entry in the EGT mean with respect to the laws of chess? With such a description in place, the problem of verifying the correctness of the EGT reduces to providing sufficient evidence that each entry in the EGT satisfies its description.

This EGT verification process was carried out in a proof-of-concept experiment that generated *correct-by-construction* four piece pawnless EGTs [17], by using the following steps.

1. The laws of chess, less Pawns and castling, were defined in higher order logic, and these definitions were entered into the HOL4 theorem prover.
2. The format of EGTs represented as ordered binary decision diagrams was also defined in HOL4.
3. For each EGT, a formal proof was constructed in the HOL4 theorem prover that all of the entries follow from the laws of chess.

The remainder of this section will briefly examine the steps of this experiment.

3.1 Formalizing the Laws of Chess

The first step of the EGT verification process involves making a precise definition of the laws of chess, by translating them from the FIDE handbook [2] into a formal logic. The Semantic Web uses description logic to describe the content of web pages, but this is not expressive enough to naturally formalise the laws of chess, and so higher order logic is chosen instead.

The precise set of definitions formalising the laws of chess are presented in a technical report [17]; to illustrate the approach it suffices to give one example. Here is an excerpt from the FIDE handbook:

Article 3.3. The rook may move to any square along the file or the rank on which it stands.

And here is the corresponding definition in higher order logic:

$$\mathsf{rookMaybeMoves}\ sq_1\ sq_2 \equiv (\mathsf{sameFile}\ sq_1\ sq_2 \vee \mathsf{sameRank}\ sq_1\ sq_2) \wedge (sq_1 \neq sq_2)$$

The definition of rook moves is completed by combining the definition of **rookMaybeMoves** with a formalisation of Article 3.5, which states that the Rook, Bishop and Queen may not move over any pieces.

In this way the whole of the laws of chess (with no pawns or castling) are formalised into about 60 definitions in higher order logic, culminating in the important

$$\mathsf{depthToMate}\ p\ n \equiv \ldots$$

EGT relation, which formalises the familiar metric from Nalimov's EGTs that position p has DTM n.

3.2 Formal Verification of EGTs

In itself the formalisation of the laws of chess into higher order logic is nothing more than a mathematical curiosity. However, matters get more interesting when the definitions are entered into an interactive theorem prover, such as the HOL4 system [50]. These theorem provers are designed with a simple logical kernel that is equipped to execute the rules of inference of the logic, and it is up to the user to break apart large proofs into a long sequence of inferences that are checked by the theorem prover. The theorem prover provides proof tools called tactics to help with the breaking apart of large proofs, but since everything must ultimately be checked by the logical kernel, the trusted part of the system is very small.[19] As a consequence of this design, a proof that can be checked by an interactive theorem prover is highly reliable. This is why it is a significant milestone that the Four Colour Theorem is now underpinned by a formal proof that has been completely checked by an interactive theorem prover [12].

How can this capability of reliable proof checking be harnessed to check EGTs? In principle, for each *(p,n)* DTM entry in an EGT, a proof of the relation **depthToMate** ***p n*** could be constructed and checked, thus establishing that the EGT entry did indeed logically follow from the laws of chess as formalised in HOL4. However, in practice the size of any EGT would make this strategy completely infeasible.

A more promising approach is to formalise the program that generates the EGT, and construct a proof that it could only generate EGT entries that followed from the laws of chess. This idea of using logic to verify formally computer programs is very old[20], but it is only recently that theorem proving technology has made it practical for significant programs, such as the Verified C Compiler [52]. This is the most promising approach for generating verified EGTs of a realistic size, but it is still currently too labour-intensive for anything less than critical infrastructure.[21]

An alternative approach works sufficiently well to construct a prototype verified EGT [17]. The whole EGT is formalised in the logic, as a sequence of sets of positions of increasing DTM. Each set of positions is encoded as a bit-vector, and stored as an *ordered binary decision diagram* or BDD [53]. Using a combination of deduction and BDD operations [54], it is possible to use the previous DTM set to construct a proof that the next set is correct. The verification is bootstrapped with the set of checkmate positions at DTM 0, which is easily checked by expanding the definition of a checkmate position.

Due to the difficulty of compressing sets of positions using BDDs [55, 56], it is only possible to generate verified EGTs for four piece pawnless endgames using this technique, but as a result there are now many win/draw/loss positions that come with a proof that they logically follow from the laws of chess [17, 57].[22]

4 Summary

This paper has examined the correctness of endgame data from multiple perspectives. A structured survey was presented on the past pattern of errors in EGTs managed during work in progress, surfacing in publications, and occurring after the data was generated. Specific remedies were proposed to counter observed classes of error from creeping in to endgame data. A particular challenge is to pin down the precise meaning of the data in an EGT, so that the reader uses the data in a context

[19] Typically, just a few hundred lines of code.

[20] The earliest reference known to the authors is a paper of Turing published in 1949 [51].

[21] It is left as a challenge to the research community to come up with a safety-critical application of EGTs.

[22] Naturally, the numbers generated by the verification agree with Nalimov's results.

that is compatible with the context in which the author computed it. A possible solution to pinning down this meaning was described that used higher order logic, and the presence of a machine-readable specification opened the door to a discussion of techniques for using interactive theorem provers to generate EGTs that are correct by construction.

Although endgame data has been the focus of the paper, the methodology of examining data assurance carries over to many opaque computations. Specifically, the learning points are that:

- it is vital to collect data on errors that have occurred in practice, to ground any discussion of data assurance,
- there is no magic solution, but rather individual remedies must be introduced that counter observed classes of error, and
- the precise meaning of the data, the exact context in which it was computed, must be encoded in some form and made available with the data, to counter misinterpretations on the part of the reader.

As society becomes increasingly dependent on computers and data generated by opaque computations, we cannot afford to overlook techniques for safeguarding data assurance.

Acknowledgments. Our review illustrates the intrinsic challenges and complexity of the computations addressed here. The fact that there are so few, largely temporary, errors is a testimony to the awareness, skill, and rigour of those who have contributed significant EGT results to date. Great achievements lead to others, e.g., the solving of Checkers in 2007 [58]. Throughout, the ICCA and ICGA encouraged and championed work on endgames. We thank all those involved for their contributions in the endgame field and for the example they have set in the field of Systems Engineering.

References

1. Hooper, D., Whyld, K.: The Oxford Companion to Chess, 2nd edn. OUP (1992)
2. FIDE: The Laws of Chess. FIDE Handbook E.1.01A (2009), `http://www.fide.com/component/-handbook/?id=124&view=article`
3. McCorduck, P.: Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence. A.K.Peters, Wellesley (2004)
4. Heinz, E.A.: Endgame databases and efficient index schemes. ICCA J. 22(1), 22–32 (1999)
5. Ströhlein, T.: Untersuchungen über kombinatorische Spiele. Ph.D. thesis, Technical University of Munich (1970)
6. Haworth, G.M^{c}C.: 6-man Chess Solved. ICGA J. 28(3), 153 (2005)
7. Kryukov, K.: EGTs Online (2007), `http://kirill-kryukov.com/chess/tablebases-online/`
8. Nalimov, E.V., Haworth, G.M^{c}C., Heinz, E.A.: Space-efficient indexing of chess endgame tables. ICGA J. 23(3), 148–162 (2000)
9. Nalimov, E.V.: Private Communications (2000)
10. Coe, T.: Inside the Pentium FDIV bug. Dr. Dobb's Journal 229, 129–135, 148 (1995)

11. Gorenstein, D., Lyons, R., Solomon, R.: The Classification of Finite Simple Groups. AMS (1994)
12. Devlin, K.: Last doubts removed about the proof of the Four Color Theorem (2005), `http://www.maa.org/devlin/devlin_01_05.html`
13. RIPE: Mediterranean Cable Cut – A RIPE NCC Analysis (2008), `http://www.ripe.net/projects/-reports/2008cable-cut/index.html`
14. BBC: Repairs begin on undersea cable (2008), `http://news.bbc.co.uk/1/hi/technology/7795320.stm`
15. Pouzzner, D.: Partial failure of Internet root nameservers. The Risks Digest, 19–25 (1997)
16. BBC: Human error' hits Google search (2009), `http://news.bbc.co.uk/1/hi/technology/7862840-.stm`
17. Hurd, J.: Formal verification of chess endgame databases. In: Hurd, J., Smith, E., Darbari, A. (eds.) Theorem proving in higher order logics: Emerging trends proceedings. Technical Report PRG-RR-05-02, 85-100. Oxford University Computing Laboratory (2005)
18. Shadbolt, N., Hall, W., Berners-Lee, T.: The Semantic Web Revisited. IEEE Intelligent Systems 21(3), 96–101 (2006)
19. Herik, H.J., van den Herschberg, I.S., Nakad, N.: A Six-Men-Endgame Database: KRP(a2)KbBP(a3). ICCA J. 10(4), 163–180 (1987)
20. Michalski, R.S., Negri, P.G.: An Experiment on Inductive Learning in Chess End Games. In: Machine Intelligence, vol. 8, pp. 175–192. Ellis Horwood (1977)
21. Sattler, R.: Further to the KRP(a2)KbBP(a3) Database. ICCA J. 11(2/3), 82–87 (1988)
22. Herik, H.J., van den Herschberg, I.S., Nakad, N.: A Reply to R. Sattler's Remarks on the KRP(a2)-KbBP(a3) Database. ICCA J. 11(2/3), 88–91 (1988)
23. Bleicher, E.: FREEZER (2009), `http://www.freezerchess.com/`
24. Andrist, R.B.: WILHELM (2009), `http://www.geocities.com/rba_schach2000/index_english.htm`
25. Wu, R., Beal, D.F.: Solving Chinese Chess Endgames by Database Construction. Information Sciences 135(3/4), 207–228 (2001)
26. Wirth, C., Nievergelt, J.: Exhaustive and Heuristic Retrograde Analysis of the KPPKP Endgame. ICCA J. 22(2), 67–80 (1999)
27. Tay, A.: A Guide to Endgame Tablebases (2009), `http://www.horizonchess.com/FAQ/Winboard/-egtb.html`
28. Merlino, J.: Regarding FEG 3.03b – List Found (2002), `http://www.horizonchess.com/FAQ/-Winboard/egdbbug.html`
29. Chessbase: FRITZ ENDGAME T3 (2006), `http://www.chessbase.com/workshop2.asp?id=3179`
30. Roycroft, A.J.: *C* Correction. EG 7(119), 771 (1996)
31. Roycroft, A.J.: The Computer Section: Correction. EG 8(123), 47–48 (1997)
32. Roycroft, A.J.: *C*. EG 8(Suppl. 130), 428 (1998)
33. Roycroft, A.J.: Snippets. EG 8(131), 476 (1999)
34. Jones, N.D., Muchnick, S.S. (eds.): TEMPO. LNCS, vol. 66. Springer, Heidelberg (1978)
35. Bourzutschky, M.S., Tamplin, J.A., Haworth, G.M^{c}C.: Chess endgames: 6-man data and strategy. Theoretical Computer Science 349(2), 140–157 (2005)
36. Bourzutschky, M.S.: Tablebase version comparison, `http://preview.tinyurl.com/d3wny4` (2006-08-10)
37. Thompson, K.: Reflections on Trusting Trust. CACM 27(8), 761–763 (1984)
38. Tamplin, J.: EGT-query service extending to 6-man pawnless endgame EGTs in DTC, DTM, DTZ and DTZ$_{50}$ metrics (2006), `http://chess.jaet.org/endings/`

39. Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., Sutphen, S.: Building the Checkers 10-piece Endgame Databases. In: Advances in Computer Games, vol. 10, pp. 193–210 (2003)
40. Thompson, K.: Retrograde Analysis of Certain Endgames. ICCA J. 9(3), 131–139 (1986)
41. Stiller, L.B.: Parallel Analysis of Certain Endgames. ICCA J. 12(2), 55–64 (1989)
42. Nunn, J.: Secrets of Pawnless Endings, 2nd Expanded edn., Gambit (2002)
43. Krabbé, T.: Private Communication (2008-09-05)
44. Herbstman, A.O.: Draw Study 172. EG 5, 195 (1967)
45. Haworth, G.M^{c}C.: Strategies for Constrained Optimisation. ICGA J. 23(1), 9–20 (2000)
46. Byers, F.R.: Care and Handling of CDs and DVDs: A Guide for Librarians and Archivists. CLIR/NIST (2003), `http://www.clir.org/pubs/reports/pub121/contents.html`
47. Fierz, M., Cash, M., Gilbert, E.: The 2002 World Computer-Checkers Championship. ICGA J. 25(3), 196–198 (2002)
48. Schaeffer, J.: One Jump Ahead: Challenging Human Supremacy in Checkers. Springer, New York (1997)
49. Komissarchik, E.A., Futer, A.L.: Ob Analize Ferzevogo Endshpilia pri Pomoshchi EVM. Problemy Kybernet 29, 211–220 (1974); Reissued in translation by Chr. Posthoff and I.S. Herschberg under the title 'Computer Analysis of a Queen Endgame. ICCA J. 9(4), 189–198 (1986)
50. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A theorem-proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
51. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, pp. 67–69. Cambridge University Mathematical Laboratory (1949)
52. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), pp. 42–54. ACM, New York (2006)
53. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (1992)
54. Gordon, M.J.C.: Programming combinations of deduction and BDD-based symbolic calculation. LMS J. of Computation and Mathematics 5, 56–76 (2002)
55. Edelkamp, S.: Symbolic exploration in two-player games: Preliminary results. In: The International Conference on AI Planning & Scheduling (AIPS), Workshop on Model Checking, Toulouse, France, pp. 40–48 (2002)
56. Kristensen, J.T.: Generation and compression of endgame tables in chess with fast random access using OBDDs. Master's thesis, U. of Aarhus, Dept. of Computer Science (2005)
57. Hurd, J.: Chess Endgames (2005), `http://www.gilith.com/chess/endgames`
58. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is Solved. Science 317(5844), 1518–1522 (2007)

Author Index