H. Jaap van den Herik
Aske Plaat (Eds.)

LNCS 7168

Advances in Computer Games

13th International Conference, ACG 2011
Tilburg, The Netherlands, November 2011
Revised Selected Papers

Springer

Lecture Notes in Computer Science 7168

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

H. Jaap van den Herik   Aske Plaat (Eds.)

Advances in Computer Games

13th International Conference, ACG 2011
Tilburg, The Netherlands, November 20-22, 2011
Revised Selected Papers

Springer

Volume Editors

H. Jaap van den Herik
Aske Plaat
Tilburg University
Tilburg Institute of Cognition and Communication
Warandelaan 2, 5037 AB Tilburg, The Netherlands
E-mail: {jaapvandenherik,aske.plaat}@gmail.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31865-8 e-ISBN 978-3-642-31866-5
DOI 10.1007/978-3-642-31866-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012941836

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

© Springer-Verlag Berlin Heidelberg 2012
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This book contains the papers of the 13th Advances in Computer Games Conference (ACG 2011) held in Tilburg, The Netherlands. The conference took place during November 20–22, 2011, in conjunction with the 16th Computer Olympiad and the 19th 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 Tilburg conference was definitively characterized by the progress of Monte Carlo Tree Search (MCTS) and the development of new games. Earlier conferences took place in London (1975), Edinburgh (1978), London (1981, 1984), Noordwijkerhout (1987), London (1990), Maastricht (1993, 1996), Paderborn (1999), Graz (2003), Taipei (2005), and Pamplona (2009).

The Program Committee (PC) was pleased to see that so much progress was made in MCTS and that on top of that new games and new techniques were added to the recorded achievements. Each paper was sent to at least three referees. If conflicting views on a paper were reported, the referees themselves arrived at an appropriate decision. With the help of many referees (see after the preface), the PC accepted 29 papers for presentation at the conference and publication in these proceedings. As usual we informed the authors that they submitted their contribution to a post-conference editing process. The two-step process is meant (1) to give authors the opportunity to include the results of the fruitful discussion after the lecture into their paper, and (2) to maintain the high-quality threshold of the ACG series. The authors enjoyed this procedure.

The above-mentioned set of 29 papers covers a wide range of computer games and many different research topics. We mention the topics in the order of publication: Monte Carlo Tree Search and its enhancements (10 papers), temporal difference learning (2 papers), optimization (4 papers), solving and searching (2 papers), analysis of a game characteristic (3 papers), new approaches (5 papers), and serious games (3 papers).

We hope that the readers will enjoy the research efforts made by the authors. Below we reproduce brief characterizations of the 29 contributions taken from the text as submitted by the authors. The authors of the first publication "Accelerated UCT and Its Application to Two-Player Games" received the Best Paper Award of ACG 2011.

"Accelerated UCT and Its Application to Two-Player Games" by Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe, and Kokolo Ikeda describes Monte-Carlo Tree Search (MCTS) as a successful approach for improving the performance of game-playing programs. A well-known weakness of MCTS is caused by the deceptive structures which often appear in game tree search. To overcome the weakness the authors present the Accelerated UCT algorithm

(Upper Confidence Bounds applied to Trees). Their finding consists in using a new back-up operator that assigns higher weights to recently visited actions, and lower weights to actions that have not been visited for a long time. Results in Othello, Havannah, and Go show that Accelerated UCT is not only more effective than previous approaches but also improves the strength of FUEGO, which is one of the best computer Go programs.

"Revisiting Move Groups in Monte Carlo Tree Search" is authored by Gabriel Van Eyck and Martin Müller. These authors also remark that the UCT (Upper Confidence Bounds applied to Trees) algorithm has been a stimulus for significant improvements in a number of games, most notably the game of Go. They investigate the use of move groups. Move groups is a modification that greatly reduces the branching factor at the cost of an increased search depth and as such it may be used to enhance the performance of UCT. From the results of the experiments, the authors arrive at a general structure of good move groups in which they determine which parameters to use for enhancing the playing strength.

In "PACHI: State-of-the-Art Open Source Go Program", Petr Baudiš and Jean-loup Gailly start describing a state-of-the-art implementation of the Monte Carlo Tree Search algorithm for the game of Go. Their PACHI software is currently one of the strongest open source Go programs, competing at the top level with other programs and playing evenly against advanced human players. The paper describes their framework (implementation and chosen algorithms) together with three notable original improvements: (1) an adaptive time control algorithm, (2) dynamic komi, and (3) usage of the criticality statistic. Moreover, new methods to achieve efficient scaling both in terms of multiple threads and multiple machines in a cluster are presented.

"Time Management for Monte Carlo Tree Search in Go" is written by Hendrik Baier and Mark H.M. Winands. So far, little has been published on time management for MCTS programs under tournament conditions. The authors investigate the effects that various time-management strategies have on the playing strength in Go. They consider strategies taken from the literature as well as newly proposed and improved ones. Moreover, they investigate both *semi-dynamic* strategies that decide about time allocation for each search before it is started, and *dynamic* strategies that influence the duration of each move search while it is already running. In their experiments, two domain-independent enhanced strategies, EARLY-C and CLOSE-N, are tested; each of them provides a significant improvement over the state of the art.

"An MCTS Program to Play EinStein Würfelt Nicht!" by Richard Lorentz describes a game that has elements of strategy, tactics, and chance. The author remarks that reasonable evaluation functions for this game can be found. Nevertheless, he constructed an MCTS program to play this game. The paper describes the basic structure and its strengths and weaknesses. Then the MCTS program is successfully compared with existing mini–max-based programs and to a pure MC version.

"Monte Carlo Tree Search Enhancements for Havannah" is authored by Jan A. Stankiewicz, Mark H.M. Winands, and Jos W.H.M. Uiterwijk. The article shows how the performance of a Monte Carlo Tree Search (MCTS) player for Havannah can be improved by guiding the search in the playout and selection steps of MCTS. To enhance the *playout* step of the MCTS algorithm, the authors used two techniques to direct the simulations, Last-Good-Reply (LGR) and N-grams. Experiments reveal that LGR gives a significant improvement, although it depends on which LGR variant is used. Using N-grams to guide the playouts also achieves a significant increase in the winning percentage. Combining N-grams with LGR leads to a small additional improvement. To enhance the *selection* step of the MCTS algorithm, the authors initialize the visit and win counts of the new nodes based on pattern knowledge. Experiments show that the best overall performance is obtained when combining the visit-and-win-count initialization with LGR and N-grams. In the best case, a winning percentage of 77.5% can be achieved against the default MCTS program.

"Playout Search for Monte-Carlo Tree Search in Multi-Player Games" by J. (Pim) A.M. Nijssen and Mark H.M. Winands proposes a technique called Playout Search. This enhancement allows the use of small searches in the playout phase of MCTS in order to improve the reliability of the playouts. The authors investigate max^n, Paranoid, and BRS for Playout Search and analyze their performance in two deterministic perfect-information multi-player games: Focus and Chinese Checkers. The experimental results show that Playout Search significantly increases the quality of the playouts in both games.

"Towards a Solution of 7x7 Go with Meta-MCTS" by Cheng-Wei Chou, Ping-Chiang Chou, Hassen Doghmen, Chang-Shing Lee, Tsan-Cheng Su, Fabien Teytaud, Olivier Teytaud, Hui-Ming Wang, Mei-Hui Wang, Li-Wen Wu, and Shi-Jim Yen is a challenging topic. So far, Go is not solved (in any sense of solving, even the weakest) beyond 6x6. The authors investigate the use of Meta-Monte-Carlo Tree Search, for building a huge 7x7 opening book. In particular, they report the 20 wins (out of 20 games) that were obtained recently in 7x7 Go against pros; they also show that in one of the games, with no human error, the pro might have won.

"MCTS Experiments on the Voronoi Game" written by Bruno Bouzy, Marc Métivier, and Damien Pellier discusses Monte Carlo Tree Search (MCTS) as a powerful tool in games with a finite branching factor. The use of MCTS on a discretization of the Voronoi game is described together with the effects of enhancements such as RAVE and Gaussian processes (GP). A set of experimental results shows that MCTS with UCB+RAVE or with UCB+GP are good first solutions for playing the Voronoi game without domain-dependent knowledge. Then the authors show how to improve the playing level by using geometrical knowledge about Voronoi diagrams, the balance of diagrams being the key concept. A new set of experimental results shows that a player using MCTS and geometrical knowledge outperforms the player without knowledge.

"4*4-Pattern and Bayesian Learning in Monte-Carlo Go" is a contribution by Jiao Wang, Shiyuan Li, Jitong Chen, Xin Wei, Huizhan Lv, and Xinhe Xu.

The authors propose a new model of pattern, namely, 4*4-Pattern, to improve MCTS in computer Go. A 4*4-Pattern provides a larger coverage and more essential information than the original 3*3-Patterns, which are currently widely used. Due to the lack of a central symmetry, it takes greater challenges to apply a 4*4-Pattern compared to a 3*3-Pattern. Many details of a 4*4-Pattern implementation are presented, including classification, multiple matching, coding sequences, and fast lookup. Additionally, Bayesian 4*4-Pattern learning is introduced, and 4*4-Pattern libraries are automatically generated from a vast amount of professional game records according to the method. The results of the experiments show that the use of 4*4-Patterns can improve MCTS in 19*19 Go to some extent, in particular when supported by 4*4-Pattern libraries generated by Bayesian learning.

"Temporal Difference Learning for Connect6" is written by I-Chen Wu, Hsin-Ti Tsai, Hung-Hsuan Lin, Yi-Shan Lin, Chieh-Min Chang, and Ping-Hung Lin. In the paper, the authors apply temporal difference (TD) learning to Connect6, and successfully use TD(0) to improve the strength of their Connect6 program, NCTU6. That program won several computer Connect6 tournaments from 2006 to 2011. The best improved version of TD learning achieves about a 58% win rate against the original NCTU6 program. The paper discusses several implementation issues that improve the program. The program has a convincing performance removing winning/losing moves via threat-space search in TD learning.

"Improving Temporal Difference Learning Performance in Backgammon Variants" by Nikolaos Papahristou and Ioannis Refanidis describes the project. Palamedes which is an ongoing project for building expert playing bots that can play backgammon variants. The paper improves upon the training method used in their previous approach for the two backgammon variants popular in Greece and neighboring countries, Plakoto and Fevga. The authors show that the proposed methods result both in faster learning as well as better performance. They also present insights into the selection of the features.

"CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning" is a contribution by Rémi Coulom. Artificial intelligence in games often leads to the problem of parameter tuning. Some heuristics may have coefficients, and they should be tuned to maximize the win rate of the program. A possible approach is to build local quadratic models of the win rate as a function of the program parameters. Many local regression algorithms have already been proposed for this task, but they are usually not sufficiently robust to deal automatically and efficiently with very noisy outputs and non-negative Hessians. The CLOP principle is a new approach to local regression that overcomes all these problems in a straightforward and efficient way. CLOP discards samples of which the estimated value is confidently inferior to the mean of all samples. Experiments demonstrate that, when the function to be optimized is smooth, this method outperforms all other tested algorithms.

"Analysis of Evaluation-Function Learning by Comparison of Sibling Nodes" written by Tomoyuki Kaneko and Kunihito Hoki, discusses gradients of search values with a parameter vector θ in an evaluation function. Recent learning

methods for evaluation functions in computer shogi are based on minimization of an objective function with search results. The gradients of the evaluation function at the leaf position of a principal variation (PV) are used to make an easy substitution of the gradients of the search result. By analyzing the variations of the min-max value, the authors show (1) when the min-max value is partially differentiable and (2) how the substitution may introduce errors. Experiments on a shogi program with about 1 million parameters show how frequently such errors occur, as well as how effective the substitutions for parameter tuning are in practice.

"Approximating Optimal Dudo Play with Fixed-Strategy Iteration Counterfactual Regret Minimization" is a contribution by Todd W. Neller and Steven Hnath. Using the bluffing dice game Dudo as a challenge domain, the authors abstract information sets by an imperfect recall of actions. Even with such abstraction, the standard Counterfactual Regret Minimization (CFR) algorithm proves impractical for Dudo, since the number of recursive visits to the same abstracted information sets increases exponentially with the depth of the game graph. By holding strategies fixed across each training iteration, the authors show how CFR training iterations may be transformed from an exponential-time recursive algorithm into a polynomial-time dynamic-programming algorithm, making computation of an approximate Nash equilibrium for the full two-player game of Dudo possible for the first time.

"The Global Landscape of Objective Functions for the Optimization of Shogi Piece Values with a Game-Tree Search" is written by Kunihito Hoki and Tomoyuki Kaneko. The landscape of an objective function for supervised learning of evaluation functions is numerically investigated for a limited number of feature variables. Despite the importance of such learning methods, the properties of the objective function are still not well known because of its complicated dependence on millions of tree-search values. The paper shows that the objective function has multiple local minima and the global minimum point indicates reasonable feature values. It is shown that an existing iterative method is able to minimize the functions from random initial values with great stability, but it has the possibility to end up with a non-reasonable local minimum point if the initial random values are far from the desired values.

"Solving Breakthrough with Race Patterns and Job-Level Proof Number Search" is a contribution by Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave. Breakthrough is a recent race-based board game usually played on an 8×8 board. The authors describe a method to solve 6×5 boards based on race patterns and an extension of the Job-Level Proof Number Search JLPNS. Using race patterns is a new domain-specific technique that allows early endgame detection. The patterns they use enable them to prune positions safely and statically as far as seven moves from the end. The authors also present an extension of the parallel algorithm (JLPNS), viz., when a PN search is used as the underlying job.

"Infinite Connect-Four Is Solved: Draw" by Yoshiaki Yamaguchi, Kazunori Yamaguchi, Tetsuro Tanaka, and Tomoyuki Kaneko, describes the newly obtained solution for variants of Connect-Four played on an infinite board. The authors proved their result by introducing never-losing strategies for both players. The strategies consist of a combination of paving patterns, which are follow-up, follow-in-CUP, and a few others. By employing the strategies, both players can block their opponents to achieve the winning condition. This means that optimal play by both players leads to a draw in these games.

"Blunder Cost in Go and Hex" is a contribution by Henry Brausen, Ryan B. Hayward, Martin Müller, Abdul Qadir, and David Spies. In Go and Hex, they examine the effect of a blunder — here, a random move — at various stages of a game. For each fixed move number, they run a self-play tournament to determine the expected blunder cost at that point.

"Understanding Distributions of Chess Performances" by Kenneth W. Regan, Bartlomiej Macieja, and Guy McC. Haworth studies the population of chess players and the distribution of their performances measured by Elo ratings and by computer analysis of moves. Evidence that ratings have remained stable since the inception of the Elo system in the 1970s is given in three forms: (1) by showing that the population of strong players fits a straightforward logistic-curve model without inflation, (2) by plotting players' average error against the FIDE category of tournaments over time, and (3) by skill parameters from a model that employs computer analysis keeping a nearly constant relation to Elo rating across that time. The distribution of the model's *intrinsic performance ratings* can therefore be used to compare populations that have limited interaction, such as between players in a national chess federation and FIDE, and to ascertain relative drift in their respective rating systems.

"Position Criticality in Chess Endgames" is a contribution by Guy McC. Haworth and Á. Rusz. In some 50,000 Win Studies in Chess, White is challenged to find an effectively unique route to a win. Judging the impact of less than absolute uniqueness requires both technical analysis and artistic judgment. Here, for the first time, an algorithm is defined to help analyze uniqueness in endgame positions objectively. The key idea is to examine how critical certain positions are to White in achieving the win. The algorithm uses sub-n-man endgame tables (EGTs) for both Chess and relevant, adjacent variants of Chess. It challenges authors of EGT generators to generalize them to create EGTs for these chess variants. The algorithm has already proved to be efficient and effective in an implementation for Starchess, itself a variant of chess. The approach also addresses a number of similar questions arising in endgame theory, games, and compositions.

"On Board-Filling Games with Random-Turn Order and Monte Carlo Perfectness" is a contribution by Ingo Althöfer. In a game, pure Monte Carlo search with parameter T means that for each feasible move T random games are generated. The move with the best average score is played. The author calls a game "Monte Carlo perfect" when this straightforward procedure converges to perfect play for each position, when T goes to infinity. Many popular games like Go, Hex,

and Amazons are NOT Monte Carlo perfect. In the paper, two-player zero-sum games are investigated where the turn-order is random: always a fair coin flip decides which player acts on the next move. A whole class of such random-turn games is proven to be Monte Carlo perfect. The result and generalizations are discussed, with example games ranging from very abstract to very concrete.

"Modeling Games with the Help of Quantified Integer Linear Programs" is written by Thorsten Ederer, Ulf Lorenz, Thomas Opfer, and Jan Wolf. Quantified linear programs (QLPs) are linear programs with mathematical variables being either existentially or universally quantified. The integer variant (Quantified linear integer program, QIP) is PSPACE-complete, and can be interpreted as a two-person zero-sum game. Additionally, it demonstrates a remarkable flexibility in polynomial reduction, such that many interesting practical problems can be elegantly modeled as QIPs. Indeed, the PSPACE-completeness guarantees that all PSPACE-complete problems, for example, games like Othello, Go-Moku, and Amazons, can be described with the help of QIPs, with only moderate overhead. The authors present the *dynamic graph reliability* (DGR) optimization problem and the game *Go-Moku* as examples.

"Computing Strong Game-Theoretic Strategies in Jotto" by Sam Ganzfried describes a new approach that computes approximate equilibrium strategies in Jotto. Jotto is quite a large two-player game of imperfect information; its game tree has many orders of magnitude more states than games previously studied, including no-limit Texas Hold'em. To address the fact that the game tree is so large, the authors propose a novel strategy representation called oracular form, in which they do not explicitly represent a strategy, but rather appeal to an oracle that quickly outputs a sample move from the strategy's distribution. Their overall approach is based on an extension of the fictitious play algorithm to this oracular setting. The authors demonstrate the superiority of their computed strategies over the strategies computed by a benchmark algorithm, both in terms of head-to-head and worst-case performance.

"Online Sparse Bandit for Card Games" is written by David L. St-Pierre, Quentin Louveaux, and Olivier Teytaud. Finding an approximation of a Nash equilibrium in matrix games is an important topic. A bandit algorithm commonly used to approximate a Nash equilibrium is EXP3. Although the solution to many problems is often sparse, EXP3 inherently fails to exploit this property. To the authors' best knowledge, there is only an offline truncation proposed to handle the sparseness issue. Therefore, the authors propose a variation of EXP3 to exploit the fact that the solution is sparse by dynamically removing arms; the resulting algorithm empirically performs better than previous versions. The authors apply the resulting algorithm to an MCTS program for the Urban Rivals card game.

"Game Tree Search with Adaptive Resolution" is authored by Hung-Jui Chang, Meng-Tsung Tsai, and Tsan-sheng Hsu. In the paper, the authors use an adaptive resolution R to enhance the min-max search with the alpha-beta pruning technique, and show that the value returned by the modified algorithm, called Negascout-with-resolution, differs from that of the original version by at

most R. Guidelines are given to explain how the resolution should be chosen to obtain the best possible outcome. The experimental results demonstrate that Negascout-with-resolution yields a significant performance improvement over the original algorithm on the domains of random trees and real game trees in Chinese chess.

"Designing Casanova: A Language for Games" is written by G. Maggiore, A. Spanó, R. Orsini, G. Costantini, M. Bugliesi, and M. Abbadi. The authors present the Casanova language, which allows the building of games with three important advantages when compared to traditional approaches: simplicity, safety, and performance. They show how to rewrite an official sample of the XNA framework, resulting in a smaller source and a higher performance.

"Affective Game Dialogues" by Michael Lankes and Thomas Mirlacher investigates natural game input devices, such as Microsoft's Kinect or Sony's Playstation Move. They have become increasingly popular and allow a direct mapping of player performance in regard to actions in the game world. Games have been developed that enable players to interact with their avatars and other game objects via gestures and/or voice input. However, current technologies and systems do not tap into the full potential of affective approaches. Affect in games can be harnessed as a supportive and easy to use input method. The paper proposes a design approach that utilizes facial expressions as an explicit input method in game dialogues. This concept allows players to interact with non-player characters (NPC) by portraying specific basic emotions.

"Generating Believable Virtual Characters Using Behavior Capture and Hidden Markov Models" by Richard Zhao and Duane Szafron proposes a method of generating natural-looking behaviors for virtual characters using a data-driven method called behavior capture. The authors describe the techniques (1) for capturing trainer-generated traces, (2) for generalizing these traces, and (3) for using the traces to generate behaviors during game-play. Hidden Markov models (HMMs) are used as one of the generalization techniques for behavior generation. The authors compared the proposed method with other existing methods by creating a scene with a set of six variations in a computer game, each using a different method for behavior generation, including their proposed method. They conducted a study in which participants watched the variations and ranked them according to a set of criteria for evaluating behaviors. The study showed that behavior capture is a viable alternative to existing manual scripting methods and that HMMs produced the most highly ranked variation with respect to overall believability.

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 Tilburg (see the beginning of this preface) have contributed substantially by bringing the researchers together. Without much emphasis, we recognize the work by the committees of the ACG 2011 as essential for this publication. One exception is made for Joke Hellemons, who is gratefully thanked for all services to our games community.

Finally, the editors happily recognize the generous sponsors Tilburg University, Tilburg Center for Cognition and Communication, LIS, ICGA, Netherlands eScience Center, NWO NCF, SURFnet, CICK, The Red Brick, NBrIX, and Digital Games Technology.

April 2012

Jaap van den Herik
Aske Plaat

Organization

Executive Committee

Editors	H. Jaap van den Herik Aske Plaat
Program Co-chairs	H. Jaap van den Herik Aske Plaat

Organizing Committee

Johanna W. Hellemons (Chair)
H. Jaap van den Herik
Erik Kamps
Aske Plaat
Susanne van der Velden
Eva Verschoor

Sponsors

Tilburg University
Tilburg Center for Cognition and Communication (TiCC)
LIS
ICGA
Netherlands eScience Center
NWO NCF
SURFnet
CICK
The Red Brick
NBrIX
Digital Games Technology

Program Committee

Ingo Althöfer
Yngvi Björnsson
Bruno Bouzy
Ivan Bratko
Tristan Cazenave
Jr-Chang Chen
Paolo Ciancarini
Rémi Coulom
Omid David Tabibi
Jeroen Donkers
David Fotland
Aviezri Fraenkel
James Glenn
Matej Guid
Dap Hartmann
Tsuyoshi Hashimoto
Guy McC. Haworth
Ryan Hayward
Tsan-Sheng Hsu
Han-Shen Huang
Hiroyuki Iida
Graham Kendall
Akihiro Kishimoto
Yoshiyuki Kotani
Clyde Kruskal
Richard Lorenz
Ulf Lorenz
Hitoshi Matsubara
John-Jules Meyer
Martin Müller
Jacques Pitrat
Christian Posthoff
Matthias Rauterberg

Alexander Sadikov
Jahn Saito
Jonathan Schaeffer
Yaron Shoham
Pieter Spronck
Nathan Sturtevant
Tetsuro Tanaka
Gerald Tesauro
Yoshimasa Tsuruoka
Jos Uiterwijk
Erik van der Werf
Peter van Emde Boas
Jan van Zanten
Gert Vriend
Hans Weigand
Mark Winands
I-Chen Wu
Georgios Yannakakis
Shi-Jim Yen

Additional Referees

Victor Allis
Darse Billings
Cameron Browne
Johannes Fürnkranz
Reijer Grimbergen
Shun-Chin Hsu
Robert Hyatt
Shaul Markovitch
Tod Neller
Pim Nijssen
Maarten Schadd
Richard Segal
David Silver
Duane Szafron
Olivier Teytaud
John Tromp
Thomas Wolf

The Advances in Computers and Chess / Games Books

The series of Advances in Computer Chess (ACC) Conferences started in 1975 as a complement to the World Computer-Chess Championships, for the first time held in Stockholm in 1974. In 1999, the title of the conference changed from ACC to ACG (Advances in Computer Games). Since 1975, 13 ACC/ACG conferences have been held. Below we list the conference places and dates together with the publication; the Springer publication is supplied with an LNCS series number.

London, England (1975, March)
Proceedings of the 1st Advances in Computer Chess Conference (ACC1)
Ed. M.R.B. Clarke
Edinburgh University Press, 118 pages.

Edinburgh, United Kingdom (1978, April)
Proceedings of the 2nd Advances in Computer Chess Conference (ACC2)
Ed. M.R.B. Clarke
Edinburgh University Press, 142 pages.

London, England (1981, April)
Proceedings of the 3rd Advances in Computer Chess Conference (ACC3)
Ed. M.R.B. Clarke
Pergamon Press, Oxford, UK,182 pages.

London, England (1984, April)
Proceedings of the 4th Advances in Computer Chess Conference (ACC4)
Ed. D.F. Beal
Pergamon Press, Oxford, UK, 197 pages.

Noordwijkerhout, The Netherlands (1987, April)
Proceedings of the 5th Advances in Computer Chess Conference (ACC5)
Ed. D.F. Beal
North Holland Publishing Comp., Amsterdam, The Netherlands, 321 pages.

London, England (1990, August)
Proceedings of the 6th Advances in Computer Chess Conference (ACC6)
Ed. D.F. Beal
Ellis Horwood, London, UK, 191 pages.

Maastricht, The Netherlands (1993, July)
Proceedings of the 7th Advances in Computer Chess Conference (ACC7)
Eds. H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk
Drukkerij Van Spijk B.V. Venlo, The Netherlands, 316 pages.

Maastricht, The Netherlands (1996, June)
Proceedings of the 8th Advances in Computer Chess Conference (ACC8)
Eds. H.J. van den Herik and J.W.H.M. Uiterwijk
Drukkerij Van Spijk B.V. Venlo, The Netherlands, 332 pages.

Paderborn, Germany (1999, June)
Proceedings of the 9th Advances in Computer Games Conference (ACG9)
Eds. H.J. van den Herik and B. Monien
Van Spijk Grafisch Bedrijf Venlo, The Netherlands, 347 pages.

Graz, Austria (2003, November)
Proceedings of the 10th Advances in Computer Games Conference (ACG10)
Eds. H.J. van den Herik, H. Iida, and E.A. Heinz
Kluwer Academic Publishers, Boston/Dordrecht/London, 382 pages.

Taipei, Taiwan (2005, September)
Proceedings of the 11th Advances in Computer Games Conference (ACG11)
Eds. H.J. van den Herik, S-C. Hsu, T-s. Hsu, and H.H.L.M. Donkers
LNCS 4250, 372 pages.

Pamplona, Spain (2009, May)
Proceedings of the 12th Advances in Computer Games Conference (ACG12)
Eds. H.J. van den Herik and P. Spronck
LNCS 6048, 231 pages.

Tilburg, The Netherlands (2011, November)
Proceedings of the 13th Advances in Computer Games Conference (ACG13)
Eds. H.J. van den Herik and A. Plaat
LNCS 7168, 356 pages.

The Computers and Games Books

The series of Computers and Games (CG) Conferences started in 1998 as a complement to the well-known series of conferences in Advances in Computer Chess (ACC). Since 1998, seven CG conferences have been held. Below we list the conference places and dates together with the Springer publication (LNCS series number).

Tsukuba, Japan (1998, November)
Proceedings of the First Computers and Games Conference (CG98)
Eds. H.J. van den Herik and H. Iida
LNCS 1558, 335 pages.

Hamamatsu, Japan (2000, October)
Proceedings of the Second Computers and Games Conference (CG2000)
Eds. T.A. Marsland and I. Frank
LNCS 2063, 442 pages.

Edmonton, Canada (2002, July)
Proceedings of the Third Computers and Games Conference (CG2002)
Eds. J. Schaeffer, M. Müller, and Y. Björnsson
LNCS 2883, 431 pages.

Ramat-Gan, Israel (2004, July)
Proceedings of the 4th Computers and Games Conference (CG2004)
Eds. H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu
LNCS 3846, 404 pages.

Turin, Italy (2006, May)
Proceedings of the 5th Computers and Games Conference (CG2006)
Eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers
LNCS 4630, 283 pages.

Beijing, China (2008, September)
Proceedings of the 6th Computers and Games Conference (CG2008)
Eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands
LNCS 5131, 275 pages.

Kanazawa, Japan (2010, September)
Proceedings of the 7th Computers and Games Conference (CG2010)
Eds. H. J. van den Herik, H. Iida, and A. Plaat
LNCS 6515, 275 pages.

Table of Contents

Accelerated UCT and Its Application to Two-Player Games 1
Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe, and Kokolo Ikeda

Revisiting Move Groups in Monte-Carlo Tree Search 13
Gabriel Van Eyck and Martin Müller

PACHI: State of the Art Open Source Go Program 24
Petr Baudiš and Jean-loup Gailly

Time Management for Monte-Carlo Tree Search in Go 39
Hendrik Baier and Mark H.M. Winands

An MCTS Program to Play EinStein Würfelt Nicht! 52
Richard J. Lorentz

Monte-Carlo Tree Search Enhancements for Havannah 60
Jan A. Stankiewicz, Mark H.M. Winands, and Jos W.H.M. Uiterwijk

Playout Search for Monte-Carlo Tree Search in Multi-player Games 72
J. (Pim) A.M. Nijssen and Mark H.M. Winands

Towards a Solution of 7x7 Go with Meta-MCTS 84
Cheng-Wei Chou, Ping-Chiang Chou, Hassen Doghmen, Chang-Shing Lee, Tsan-Cheng Su, Fabien Teytaud, Olivier Teytaud, Hui-Ming Wang, Mei-Hui Wang, Li-Wen Wu, and Shi-Jim Yen

MCTS Experiments on the Voronoi Game 96
Bruno Bouzy, Marc Métivier, and Damien Pellier

4*4-Pattern and Bayesian Learning in Monte-Carlo Go 108
Jiao Wang, Shiyuan Li, Jitong Chen, Xin Wei, Huizhan Lv, and Xinhe Xu

Temporal Difference Learning for Connect6 121
I-Chen Wu, Hsin-Ti Tsai, Hung-Hsuan Lin, Yi-Shan Lin, Chieh-Min Chang, and Ping-Hung Lin

Improving Temporal Difference Learning Performance in Backgammon Variants 134
Nikolaos Papahristou and Ioannis Refanidis

CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning 146
Rémi Coulom

Analysis of Evaluation-Function Learning by Comparison of Sibling Nodes 158
Tomoyuki Kaneko and Kunihito Hoki

Approximating Optimal Dudo Play with Fixed-Strategy Iteration Counterfactual Regret Minimization 170
Todd W. Neller and Steven Hnath

The Global Landscape of Objective Functions for the Optimization of Shogi Piece Values with a Game-Tree Search 184
Kunihito Hoki and Tomoyuki Kaneko

Solving BREAKTHROUGH with Race Patterns and Job-Level Proof Number Search 196
Abdallah Saffidine, Nicolas Jouandeau, and Tristan Cazenave

Infinite Connect-Four Is Solved: Draw 208
Yoshiaki Yamaguchi, Kazunori Yamaguchi, Tetsuro Tanaka, and Tomoyuki Kaneko

Blunder Cost in Go and Hex 220
Henry Brausen, Ryan B. Hayward, Martin Müller, Abdul Qadir, and David Spies

Understanding Distributions of Chess Performances 230
Kenneth W. Regan, Bartlomiej Macieja, and Guy McC. Haworth

Position Criticality in Chess Endgames 244
Guy McC. Haworth and Á. Rusz

On Board-Filling Games with Random-Turn Order and Monte Carlo Perfectness 258
Ingo Althöfer

Modeling Games with the Help of Quantified Integer Linear Programs 270
Thorsten Ederer, Ulf Lorenz, Thomas Opfer, and Jan Wolf

Computing Strong Game-Theoretic Strategies in Jotto 282
Sam Ganzfried

Online Sparse Bandit for Card Games 295
David L. St-Pierre, Quentin Louveaux, and Olivier Teytaud

Game Tree Search with Adaptive Resolution 306
Hung-Jui Chang, Meng-Tsung Tsai, and Tsan-sheng Hsu

Designing Casanova: A Language for Games 320
Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Giulia Costantini, Michele Bugliesi, and Mohamed Abbadi

Affective Game Dialogues: Using Affect as an Explicit Input Method in Game Dialogue Systems 333
Michael Lankes and Thomas Mirlacher

Generating Believable Virtual Characters Using Behavior Capture and Hidden Markov Models 342
Richard Zhao and Duane Szafron

Author Index 355

Accelerated UCT and Its Application to Two-Player Games

Junichi Hashimoto[1,2], Akihiro Kishimoto[3], Kazuki Yoshizoe[4], and Kokolo Ikeda[1]

[1] Japan Advanced Institute of Science and Technology
[2] Tilburg center for Cognition and Communication
[3] Tokyo Institute of Technology and Japan Science and Technology Agency
[4] The University of Tokyo

Abstract. Monte-Carlo Tree Search (MCTS) is a successful approach for improving the performance of game-playing programs. This paper presents the Accelerated UCT algorithm, which overcomes a weakness of MCTS caused by deceptive structures which often appear in game tree search. It consists in using a new backup operator that assigns higher weights to recently visited actions, and lower weights to actions that have not been visited for a long time. Results in Othello, Havannah, and Go show that Accelerated UCT is not only more effective than previous approaches but also improves the strength of FUEGO, which is one of the best computer Go programs.

1 Introduction

MCTS has achieved the most remarkable success in developing strong computer Go programs [6,7,10], since traditional minimax-based algorithms do not work well due to a difficulty in accurately evaluating positions. The Upper Confidence bound applied to Trees (UCT) algorithm [13] is a well-known representative of MCTS. It is applied to Go and many other games such as Amazons [12,15] and Havannah [19].

MCTS consists of two procedures, the Monte-Carlo simulation called *playout*, and the tree search. In a playout at position P, each player keeps playing a randomly selected move until reaching a terminal position. The outcome of the playout at P is defined as o. The outcome o (e.g., win, loss, or draw) of a terminal position is defined by the rule of the game. In the tree-search procedure, each move contains a value indicating the importance of selecting that move. For example, UCT uses the Upper Confidence Bound (UCB) value [1] (explained later) as such a criterion.

MCTS repeats the following steps until it is time to play a move. First, starting at the root node, MCTS traverses the current tree in a best-first manner by selecting the most promising move until reaching a leaf node. Then, if the number of visits reaches a pre-determined threshold, the leaf is expanded to build a larger tree. Next, MCTS performs a playout at the leaf to calculate its outcome. Finally,

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 1–12, 2012.
© Springer-Verlag Berlin Heidelberg 2012

MCTS traces back along the path from the leaf to the root and update the values of all the affected moves based on the playout result at the leaf.

At each internal node n, UCT selects move j with the largest UCB value defined as:

$$\mathrm{ucb}_j := r_j + C\sqrt{\frac{\log s}{n_j}}, \tag{1}$$

where r_j is the winning ratio of move j, s is the number of visits to n, n_j is the number of visits to j, and C is a constant value that is empirically determined.

The inaccuracy of the winning ratio is measured by the second term, the *bias term*, which decreases as the number of playouts increases. Given an infinite number of playouts, selecting the move with the highest winning ratio is proved to be optimal. However, the transitory period in which UCT chooses the suboptimal move(s) can last for a very long time [4] due to the over-optimistic behavior of UCT.

One way to overcome the above situation is to discount wins and losses for the playouts performed previously. Because the current playout is performed at a more valuable leaf than previously preformed ones, the current playout result is considered to be more reliable. Thus, MCTS can search a more important part of the search tree by forgetting the side effects of previous playouts. Although Kocsis and Szepesvári's Discounted UCB algorithm [14] is an example of such an approach, results for adapting it to tree search have not been reported yet except for an unsuccessful report in the Computer Go Mailing List [8].

This paper introduces a different way of focusing on more recent winning ratios than Discounted UCB. Its contributions are threefold.

1. Development of the *Accelerated UCT* algorithm that maintains the reliability of past winning ratios and focuses on exploring the subtrees where playout results are recently backed up.
2. Experimental results showing the potential of Accelerated UCT over plain UCT and Discounted UCT, which is an application of Discounted UCB to tree search, in Othello, Havannah, and Go.
3. Experimental results clearly showing that Accelerated UCT with Rapid Action Value Estimate [9,20] further improves the strength of FUEGO [7], a strong computer Go program that is freely available.

The structure of the paper is as follows: Section 2 explains a drawback of UCT. Section 3 reviews the literature. Section 4 describes Accelerated UCT, followed by experimental results in Section 5. Section 6 discusses conclusions and future work.

2 Drawback of UCT

Kocsis and Szepesvári proved that the UCB value converges to the optimal value [13]. This indicates that UCT can eventually select a move identical to

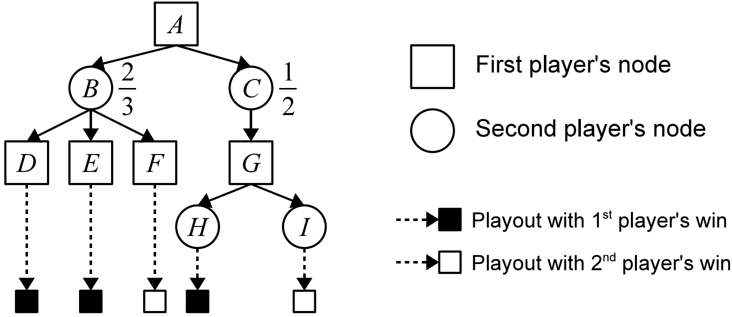


Fig. 1. An example of a deceptive structure which leads UCT to select a losing move even if playout results are accurate. The winning ratio of the first player is shown.

the one selected by the minimax (i.e., optimal) strategy. However, while this theoretical property assumes a condition in which unlimited time is given to UCT, UCT must determine the next move to play in *limited time*, that is, the number of playouts UCT can perform in practice is usually not sufficiently large to converge to the optimal value. As discussed in details below, UCT therefore suffers from a *deceptive structure* in its partially built tree, which may be a cause of selecting a wrong move that may lead to losing a game.

Because UCT's playouts often mistakenly evaluate winning positions as losses or vice versa, one example of deceptive structures is caused by an inaccurate winning ratio computed by such faulty playout results. Playout policies could be modified to decrease the frequency of encountering the deceptive structures with knowledge-based patterns (e.g., [5,10]).

However, even if playout results are accurate, UCT may still suffer from deceptive structures in the currently built tree. This drawback of UCT is explained with the help of Fig. 1. Assume that D, E, and H are winning positions and F and I are losing positions for the first player. Then, $A \to B$ and $A \to C$ are losing and winning moves with the optimal strategy, respectively, because $A \to B \to F$ is a loss and $A \to C \to G \to H$ is a win. Also assume that UCT has currently performed only one playout at each leaf in the tree built as in this figure[1]. If the playout results at D, E, and H are wins and the playout results at F and I are losses, the winning ratio of $A \to B$ is larger than that of $A \to C$ (2/3 versus 1/2), resulting in UCT choosing a losing move. UCT can obviously calculate the more accurate winning ratio by visiting B more frequently. However, UCT remains to be *deceived* to select $A \to B$ as a more promising move than $A \to C$ until $A \to B$ turns out to be valueless.

In general, deceptive structures tends to appear when MCTS must select the best move at a node with only a few promising children (e.g., ladders in Go).

[1] For simplicity, we also assume that C in equation 1 is very small (close to 0), although the drawback of UCT occurs even with large C.

3 Related Work

A number of approaches have been presented to correct the deceived behavior of MCTS in the literature, including modified playout policies (e.g., [5,10,11,17]). Although modifying playout policies is presumed to decrease deceptive structures in the search tree, these techniques are mostly based on domain-dependent knowledge. Additionally, this approach cannot completely remove the deceptive structures as shown in Fig. 1. In contrast, we aim at avoiding the deceptive structures in a domain-independent way.

The rest of this section deals with related work based on different formulas than UCB to bypass such deceptions. The approaches are orthogonal to modifying playout policies and can be usually combined with these policies.

Coquelin and Munos showed an example in which UCT works poorly due to its over-optimistic behavior if it performs only a small number of playouts [4]. They presented the Bandit Algorithm for Smooth Trees (BAST) to overcome this issue by modifying the bias term of the UCB formula. A theoretical property was proved about the regret bound and the playout sizes performed on suboptimal branches, when their smoothness assumption is satisfied in the tree. Since they did not show empirical results in games, it is still an open question whether BAST is effective for two player games or not.

Discounted UCB [14] gradually forgets past playout results to value more recent playout results. It introduces the notion of *decay*, which was traditionally presented in temporal difference learning [18]. The original UCB value is modified to the Discounted UCB value as explained below.

In the multi-armed bandit problem, let $r^D_{t,j}$ and $n^D_{t,j}$ be the discounted winning ratio of branch j and the discounted visits to j, respectively, after the t-th playout is performed. The Discounted UCB value $\mathrm{ducb}_{t,j}$ is defined as[2]:

$$\mathrm{ducb}_{t,j} := r^D_{t,j} + C\sqrt{\frac{\log s^D_t}{n^D_{t,j}}}, \tag{2}$$

where $s^D_t := \sum_i n^D_{t,i}$ and C is a constant. Discounted UCB incrementally updates $r^D_{t,j}$ and $n^D_{t,j}$ in the following way:

$$n^{\mathrm{D}}_{t+1,j} \leftarrow \lambda \cdot n^{\mathrm{D}}_{t,j} + \mathbb{I}(t+1, j), \tag{3}$$

$$r^{\mathrm{D}}_{t+1,j} \leftarrow \left(\lambda \cdot n^{\mathrm{D}}_{t,j} \cdot r_{t,j} + \mathbb{R}(t+1, j)\right) / \left(\lambda \cdot n^{\mathrm{D}}_{t,j} + \mathbb{I}(t+1, j)\right), \tag{4}$$

where λ is a constant value of decay ranging $(0, 1]$, $\mathbb{I}(t, j)$ is set to 1 if j is selected at the t-th playout or 0 otherwise. $\mathbb{R}(t, j)$ is defined as the result of the t-th playout (0 or 1 in this paper for the sake of simplicity) at j if j is selected,

[2] Precisely, Kocsis and Szepesvári used the UCB1-Tuned formula [10] to define the Discounted UCB algorithm. However, we use the standard UCB1 formula here, since we believe that this difference does not affect the properties of Discounted UCB.

or 0 if j is not selected. We assume $r^D_{0,j} = n^D_{0,j} = 0$ for any j but $\text{ducb}_{0,j}$ has a very large value so that j can be selected at least once. Note that Discounted UCB is identical to UCB if $\lambda = 1.0$. Additionally, selecting the right λ plays an important role in Discounted UCB's performance.

Discounted UCB selects branch j with the largest discounted UCB value and performs a playout at j, and then updates the discounted UCB values of *all* the branches. In other words, while Discounted UCB updates $\text{ducb}_{t,j}$ for selected branch j, the Discounted UCB values for the other unselected branches are also discounted. Discounted UCB repeats these steps until it performs sufficiently many playouts to select the best branch.

While Discounted UCB could be applied to tree search in a similar manner to UCT, one issue must be addressed (see Subsection 5.1 for details) and no success in combining Discounted UCB with tree search has been reported yet.

Ramanujan and Selman's UCTMAX_H combines UCT and minimax tree search [16]. It simply replaces performing a playout with calling an evaluation function at each leaf and backs up its minimax value instead of the mean value. Although they showed that UCTMAX_H outperforms UCT and minimax search in the game of Mancala, their approach is currently limited to domains where both minimax search and UCT perform well.

Other related work includes approaches using other algorithms as baselines rather than UCT and their applicability to UCT remains an open question. For example, instead of computing the winning ratio, Coulom introduced the "Mix" operator that is a linear combination of robust max and mean [6].

4 Accelerated UCT

Our Accelerated UCT algorithm aims at accelerating to search in a direction for avoiding deceptive structures in the partially built tree. Similarly to Discounted UCB, Accelerated UCT considers recent playouts to be more valuable. However, unlike the decay of Discounted UCB, Accelerated UCT non-uniformly decreases the reliability of subtrees that contain the past playout results.

As in UCT, Accelerated UCT keeps selecting the move with the largest *Accelerated UCB value* from the root until reaching a leaf. The Accelerated UCB value aucb_j for move j is defined as:

$$\text{aucb}_j := r^{\text{A}}_j + C\sqrt{\frac{\log s}{n_j}}, \tag{5}$$

where the bias term is identical to UCB and r^{A}_j is the *accelerated winning ratio* (explained later) defined by the notion of *velocity*. If Accelerated UCT is currently at position P for the t-th time, the velocity $v_{t,j}$ of *each of* the legal moves j at P is updated in the following way:

$$v_{t+1,j} \leftarrow v_{t,j} \cdot \lambda + \mathbb{I}(t+1, j), \tag{6}$$

where $\mathbb{I}(t+1,j)$ is 1 if move j is selected at P and is 0 otherwise, and λ is a decay ranging $(0,1]$, which has a similar effect to the decay of Discounted UCB and is empirically preset. For any move j, $v_{0,j}$ is set to 0.

Let r_i^{A} be the accelerated winning ratio of move i that creates position P and r_j^{A} be the accelerated winning ratio of move j at P. When Accelerated UCT backs up a playout result, it updates r_i^{A} with the following velocity-based weighted sum of r_j^{A}:

$$r_i^{\mathrm{A}} := \sum_{j \in LM(P)} w_j \cdot r_j^{\mathrm{A}}, \tag{7}$$

where $LM(P)$ is all the legal moves at P and $w_j := v_{t,j} / (\sum_{k \in LM(P)} v_{t,k})$.

If move j is selected, $v_{t,j}$ and w_j are increased, resulting in giving r_j^{A} a heavier weight. Additionally, Accelerated UCT is identical to UCT if $\lambda = 1$.

When a leaf is expanded after performing several playouts, Accelerated UCT must consider a way of reusing these playout results. We prepare an additional child called *virtual child* for this purpose.

Fig. 2 illustrates the virtual child of node a, represented as a'. Nodes a, b and c are leaves in the left figure. Assume that three playouts are performed at a and the winning ratio of move $r \to a$ is 2/3. Then, if a is expanded, Accelerated UCT generates two *real* children (i.e., d and e) and one virtual child a' as shown in the right figure. Additionally, assume that one playout per real child is performed (the playout results are a win at d and a loss at e).

In case of plain UCT, the winning ratio of $r \to a$ can be easily calculated as 3/5 in this situation, because the playout results previously performed at a (i.e.,the winning ratio of 2/3) are accumulated at $r \to a$. However, Accelerated UCB updates the accelerated winning ratio $r_{r \to a}^{\mathrm{A}}$ of $r \to a$ based on the winning ratios of $a \to d$ and $a \to e$ in the original definition. These moves do not include

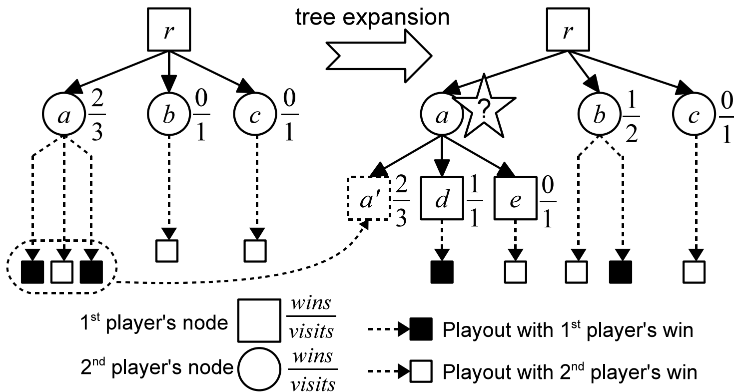


Fig. 2. An example illustrating the necessity of virtual child in Accelerated UCT

the value of 2/3. We therefore add one virtual child a' and (virtual) move $a \to a'$ to set the winning ratio of $a \to a'$ to 2/3. That is, as if there were three moves ($a \to a'$, $a \to d$ and $a \to e$) at a, Accelerated UCT updates $r^{\mathrm{A}}_{r \to a}$.

Note that the velocity of a' is also considered but is always decayed because a' is never selected.

5 Experiments

Below we describe our experiments by providing implementation details (in 5.1), outlining the setup (in 5.2), and giving two performance comparisons (in 5.3 and 5.4).

5.1 Implementation Details

We implemented the plain UCT, Discounted UCT (Discounted UCB plus tree search), and Accelerated UCT algorithms to evaluate the performance of these algorithms in the games of Othello, Havannah, and Go. All the algorithms were implemented on top of FUEGO 0.4.1[3] in Go. In contrast, we implemented them from scratch in Othello and Havannah.

Since Discounted UCB updates all the branches of the root in the multi-armed bandit problem, one possible Discounted UCT implementation is to update all the Discounted UCB values in the currently built tree. However, because this approach obviously incurs non-negligible overhead, our implementation updates the Discount UCB values in the same way as Accelerated UCT updates velocities. In this way, our Discounted UCT implementation can recompute the Discounted UCB values of "important" moves with a small overhead.

5.2 Setup

Experiments were performed on a dual hexa-core Xeon X5680 machine with 24 GB memory. While this machine has 12 cores in total, we used a single core to run each algorithm with sufficient memory.

We set the limit of the playout size to 50,000 when each algorithm determined the move to play. Although Discounted/Accelerated UCT requires an extra overhead to compute Discounted/Accelerated UCB values compared to plain UCT, we observed that this overhead was negligible.

We held a 1000-game match to compute the winning percentage for each algorithm in each domain. A draw was considered to be a half win when the winning percentage was calculated. The ratio of draws to the total number of games ranged 0.9–2.2% in Havannah, while this number was at most 7.8% in Othello. The games results were always either wins or losses in 9×9 Go with

[3] The source code is available at http://fuego.sourceforge.net/. The latest implementation is version 1.1. However, version 0.4.1 was the latest one when we started implementing the aforementioned algorithms.

7.5 komi. Because MCTS has randomness for playout results, we disabled the opening book for the experiments and thus obtained a variety of games per match.

5.3 Performance Comparison of the Plain, Discounted, and Accelerated UCT Algorithms

Table 1 shows the winning percentages of Accelerated/Discounted UCT with various decays against plain UCT in Othello, Havannah, and 9×9 Go with 95% confidence intervals, calculated by $2\sqrt{p(100-p)/1000}$, where p is the winning percentage. The best results are marked by bold text. We varied decay $\lambda = 1 - 0.1^k$ $(1 \leq k \leq 7)$ for both Discounted and Accelerated UCT. For example, $\lambda = 0.9999999$ with $k = 7$ and $\lambda = 0.9$ with $k = 1$. This implies that λ becomes close to 1 more extremely with larger k, resulting in Discounted/Accelerated UCT behaving more similarly to plain UCT.

FUEGO's default policy that performs a smarter playout based on game-dependent knowledge was used in the experiments in Go. Additionally other important enhancements except Rapid Action Value Estimate (RAVE)[4] [9] were turned on there. In contrast, in Havannah and Othello, when a playout was performed, one of the legal moves was selected with uniform randomness without incorporating any domain-specific knowledge. Additionally, no techniques enhancing the performance of UCT variants were incorporated there. C was set to 1.0 in all the domains.

The winning percentages in the table indicate that Accelerated UCT was consistently better than Discounted UCT. There is at least one case of k where Accelerated UCT was statistically superior to plain UCT in each domain, although the best k depends on the domain. In contrast, even with the best k, Discounted UCT was at most as strong as plain UCT, implying the importance of introducing a new way of decaying the winning ratio which is different from the Discounted UCB value.

Table 1. Performance comparison

(a) Accelerated vs plain UCT

	Winning percentage (%)		
k	Othello	Havannah	Go
1	39.2 ± 3.1	24.2 ± 2.7	0.8 ± 0.6
2	52.5 ± 3.2	25.1 ± 2.7	41.3 ± 3.1
3	$\mathbf{56.3 \pm 3.2}$	47.8 ± 3.2	54.5 ± 3.1
4	52.7 ± 3.1	$\mathbf{58.2 \pm 3.1}$	$\mathbf{56.0 \pm 3.1}$
5	51.3 ± 3.2	50.6 ± 3.2	53.0 ± 3.2
6	47.4 ± 3.2	48.0 ± 3.2	51.0 ± 3.2
7	48.2 ± 3.2	50.2 ± 3.2	50.1 ± 3.2

(b) Discounted vs plain UCT

	Winning percentage (%)		
k	Othello	Havannah	Go
1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
2	1.3 ± 0.7	0.0 ± 0.0	0.0 ± 0.0
3	28.0 ± 2.8	8.0 ± 1.7	0.0 ± 0.0
4	47.5 ± 3.2	31.3 ± 2.9	18.6 ± 2.5
5	49.0 ± 3.2	48.5 ± 3.2	45.8 ± 3.2
6	$\mathbf{49.2 \pm 3.2}$	$\mathbf{52.0 \pm 3.2}$	46.6 ± 3.2
7	48.2 ± 3.2	50.0 ± 3.2	$\mathbf{48.7 \pm 3.2}$

[4] We intend to show the potential of the Accelerated and Discounted UCT algorithms against UCT in this subsection. See the performance comparison with turning on RAVE in the next subsection.

The constant value of C may impact the performance of Discounted UCT since the best C might be different from plain and Accelerated UCT due to the different formula of the biased term in Discounted UCT. However, we did not exploit the best C for plain and Accelerated UCT either. Moreover, we verified that the values of the biased terms of Discounted and Accelerated UCT with the best λ were quite similar when we ran these algorithms with many positions.

Despite inclusions of FUEGO's essential enhancements to improve its playing strength except RAVE, Accelerated UCT still achieved non-negligible improvement. Additionally, even if no enhancements were incorporated in Othello and Havannah, Accelerated UCT was better than plain UCT. These results indicate that Accelerated UCT was able to remedy the deceived behavior of UCT which could not be corrected completely by the enhancements presented in the previous literature (e.g., modifications to playout policies).

If the winning ratio was over-discounted (i.e., in case of small k), both Discounted and Accelerated UCT performed poorly. However, Accelerated UCT was still more robust than Discounted UCT to the change of λ (see Table 1 again). In the extreme case of $k = 1$ where $\lambda = 0.9$, we observed that Discounted UCT won no games against plain UCT. This result indicates that Discounted UCT suffers from undesirable side effects if it eventually forgets all the past playout results that often contain useful information. In contrast, Accelerated UCT often bypasses this drawback of Discounted UCT, because the backup rule of Accelerated UCT still takes into account the past valuable playout results.

5.4 Performance Comparison with RAVE in Go

The RAVE enhancement [9] plays a crucial role in drastically improving the strength of many computer Go programs including FUEGO. One question is how to combine Discounted or Accelerated UCT with RAVE. This subsection answers the question and shows experimental results when RAVE is turned on in FUEGO, that is, we used the best configuration of FUEGO as a baseline.

RAVE assumes that there is a strong correlation between the result of a playout and the moves that appear during performing that playout as in [2,3]. RAVE then sets the playout result as the value of these moves (we call this value the *RAVE playout value*) so that the UCB values of the moves (called the *RAVE values* precisely) can be updated with their RAVE playout values even at different positions. While the original RAVE formula appears in [9], FUEGO uses a slightly different formula in [20]. The RAVE value of move j (rave_j) is defined as[5]:

$$\text{rave}_j := \frac{n_j}{n_j + W_j} r_j + \frac{W_j}{n_j + W_j} r_j^{\text{RAVE}} \quad (8)$$

[5] The RAVE value could have a bias term as in the UCB value. However, it is omitted in many computer Go programs in practice because the second term of rave_j often has a similar effect to the bias term. The bias term was not therefore included in the experiments here since FUEGO also performs best with no bias term.

where r_j is the winning ratio of j, n_j is the number of visits to j, r_j^{RAVE} is the RAVE playout value of j, and W_j is the unnormalized weight of the RAVE estimator (see [20] for details). Instead of using the UCB value, FUEGO keeps selecting move j with the highest rave_j from the root until reaching a leaf to perform a playout.

RAVE tries to converge empirically to the game value more quickly than UCT, which is successful in current Go programs. As a result, this property might have a complementary effect on avoiding deceptive structures in the tree.

In our Discounted UCT implementation, r_j and n_j in rave_j are replaced by $r_{t,j}^{\mathrm{D}}$ and $n_{t,j}^{\mathrm{D}}$ in Equations 3 and 4, respectively[6]. In contrast, in our Accelerated UCT implementation, only r_j is replaced by r_j^{A} in Equation 7.

Table 2. Performance comparison with FUEGO with switching on RAVE and all important enhancements in Go

(a) Accelerated vs FUEGO

k	Winning percentage (%)
1	51.2 ± 3.2
2	51.9 ± 3.3
3	54.5 ± 3.4
4	$\mathbf{55.9 \pm 3.5}$
5	53.8 ± 3.4
6	54.6 ± 3.4

(b) Discounted vs FUEGO

k	Winning percentage (%)
1	0.0 ± 0.0
2	0.0 ± 0.0
3	4.4 ± 0.3
4	41.6 ± 2.6
5	47.9 ± 3.0
6	$\mathbf{49.8 \pm 3.1}$

Table 2 shows the winning percentages of Discounted and Accelerated UCT with RAVE against FUEGO with the best configuration which also includes RAVE. Accelerated UCT statistically performs better than FUEGO, which implies that RAVE does not always fix the problem of deceptions in MCTS and Accelerated UCT may correct some of the deceived behaviors of MCTS. In the best case, the winning percentage of Accelerated UCT against FUEGO was 55.9 % if λ is set to 0.9999. In contrast, Discounted UCT was again at most as strong as FUEGO, as we saw similar tendencies in the previous subsection. Discounted UCT lost all the games if the winning ratio is over-discounted with small k (i.e., $k \leq 2$), while Accelerated UCT was very robust to the change of k. Again, this indicates that Discounted UCT inherently has a side-effect of forgetting past valuable playout results.

6 Concluding Remarks

We have developed the Accelerated UCT algorithm that avoids some of the deceptions that appear in the search tree of MCTS. Our experimental results have shown that Accelerated UCT not only outperformed plain and Discounted

[6] Unlike in the definition of Discounted UCB, t and j indicate the situation after the tth update *for move* j is performed.

UCT in a variety of games but also contributed to improving the playing strength of Fuego, which is one of the best computer Go programs.

Although Accelerated UCT is shown to be promising, the most important future work is to develop a technique that automatically finds a reasonable value of decay λ. At present, we must try to find a good value of λ empirically by hand. In our experiments, the best λ depends on the target domain. Additionally, since we believe that the best λ also depends on the time limit, it would be necessary for Accelerated UCT to change automatically the value of λ, based on a few factors such as the shape of the current search tree.

Acknowledgments. This research was supported by the JST PRESTO program. We thank Tomoyuki Kaneko and Martin Müller for their valuable comments on the paper.

References

1. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning 47(2-3), 235–256 (2002)
2. Bouzy, B., Helmstetter, B.: Monte Carlo Go developments. In: Proc. of the 10th International Conference on Advances in Computer Games (ACG 2010). IFIP, vol. 263, pp. 159–174. Kluwer Academic (2003)
3. Brügmann, B.: Monte Carlo Go (1993), http://www.ideanest.com/vegos/MonteCarloGo.pdf
4. Coquelin, P.-A., Munos, R.: Bandit algorithms for tree search. In: Proc. of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI 2007), pp. 67–74. AUAI press (2007)
5. Coulom, R.: Computing Elo ratings of move patterns in the game of Go. ICGA Journal 30(4), 198–208 (2007)
6. 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)
7. Enzenberger, M., Müller, M., Arneson, B., Segal, R.: Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 259–270 (2010)
8. Gelly, S.: Discounted UCB. Posted to Computer Go Mailing List (2007), http://www.mail-archive.com/computer-go@computer-go.org/msg02124.html
9. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proc. of the 24th International Conference on Machine Learning (ICML 2007). ACM International Conference Proceeding Series, vol. 227, pp. 273–280 (2007)
10. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo Go. Technical Report RR-6062, INRIA (2006)
11. Huang, S.-C., Coulom, R., Lin, S.-S.: Monte-Carlo Simulation Balancing in Practice. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 81–92. Springer, Heidelberg (2011)
12. Kloetzer, J., Iida, H., Bouzy, B.: A comparative study of solvers in Amazons endgames. In: Proc. of the IEEE Symposium on Computational Intelligence and Games (CIG 2008), pp. 378–384. IEEE Press (2008)

13. 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)
14. Kocsis, L., Szepesvári, C.: Discounted UCB. Video Lecture. In: The Lectures of PASCAL Second Challenges Workshop (2006), Slides are available at http://www.lri.fr/~sebag/Slides/Venice/Kocsis.pdf. Video is available at http://videolectures.net/pcw06_venice/
15. 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)
16. Ramanujan, R., Selman, B.: Trade-offs in sampling-based adversarial planning. In: Proc. of 21st International Conference on Automated Planning and Scheduling (ICAPS 2011), pp. 202–209. AAAI (2011)
17. Silver, D., Tesauro, G.: Monte-Carlo simulation balancing. In: Proc. of the 26th International Conference on Machine Learning (ICML 2009). ACM International Conference Proceeding Series, vol. 382, pp. 945–952 (2009)
18. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine Learning 3(1), 9–44 (1988)
19. Teytaud, F., Teytaud, O.: Creating an Upper-Confidence-Tree Program for Havannah. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 65–74. Springer, Heidelberg (2010)
20. Tom, D., Müller, M.: A Study of UCT and Its Enhancements in an Artificial Game. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 55–64. Springer, Heidelberg (2010)

Revisiting Move Groups in Monte-Carlo Tree Search

Gabriel Van Eyck and Martin Müller

University of Alberta, Edmonton, Canada
{vaneyck,mmueller}@ualberta.ca

Abstract. The UCT (Upper Confidence Bounds applied to Trees) algorithm has allowed for significant improvements in a number of games, most notably the game of Go. Move groups is a modification that greatly reduces the branching factor at the cost of increased search depth and as such may be used to enhance the performance of UCT. From the results of the experiments, we arrive at the general structure of good move groups and the parameters to use for enhancing the playing strength.

1 Introduction

Using Monte-Carlo search as a basis, the UCT algorithm has greatly improved the computer programs for many games, including all state of the art Go programs, such as Fuego [8], and some Amazons programs, such as Invader [7]. The UCT algorithm provides a method of selecting the next node to simulate in tree searches and has proven to be an effective way of guiding the large number of simulations that Monte-Carlo search uses.

However, in several games, the complexity resulting from the high branching factor makes it difficult to find the few good moves among thousands. Notably, Amazons on a tournament-sized board initially has a branching factor of 2176 and an average branching factor of approximately 500 [7]. Enhancements implemented to handle this problem tend to be game specific in nature. One potential refinement is the UCT algorithm's default policy for first-play urgency (FPU). First-play urgency is a value representing how soon to simulate a node for the first time. In basic UCT, the value is infinity; all nodes are simulated once before any further simulations [6]. In the game of Go, this value has been adjusted, based on the playing performance to achieve a greater playing strength by, Wang and Gelly [10].

Rapid Action Value Estimation (RAVE) is a second refinement of UCT. RAVE has been used in many Go programs [5], including Fuego, MoGo, and Erica, to great successes. Using RAVE modifies the original UCT algorithm to include an additional term that is based on statistics gathered for playing a stone at any point in the game, rather than just considering the next move.

When looking for enhancements that are not game-specific, an example is iterative widening. With iterative widening, the goal is to begin searching in a subset of possible moves, then gradually widening the search window if time allows. This has been used to improve Monte-Carlo tree search by Chaslot et al. [3].

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 13–23, 2012.
© Springer-Verlag Berlin Heidelberg 2012

The enhanced algorithm won significantly more often with iterative widening, or Progressive Unpruning as they called it, turned on. Against GnuGo, they won 58% of games with iterative widening and another strategy turned on and only 25% with both turned off. Prior knowledge in the form of a pattern database was used in this case, but there is potential for less game-specific methods of selecting the move subset, such as using a RAVE value.

A third general strategy would be incorporating move groups into the game tree. A move group is a selection of available moves to introduce another layer in the tree. A player's turn is split into phases: first, select a move group, and then, select a move within that group. This method has been used in Amazons and Go. In Amazons, the largest gain is in move generation savings, as shown by Richard Lorentz [7]. In his program INVADER, a turn consists of selecting an amazon, selecting a move for that amazon, then selecting a destination for its arrow. He notes that this saves time on the move generation itself by a factor of 10 simply because it does not have to generate all possible moves. For Go, Childs et al. [4] were able to increase the playing strength of their modified libEGO program by grouping moves according to their Manhattan distance from both of the previous moves. In this case the groups were not disjoint, unlike in the Amazons example. An earlier Go article by Saito et al. used disjoint groups composed first of moves within two points of the last move, then moves on the border of the game board, and lastly a group with all other moves [9]. Besides finding an increase in playing performance, they note that their player seems to shift focus better and play non-local moves when appropriate.

Given the need for a general enhancement when working with UCT, this paper explores move groups in detail. After analyzing the results from several experiments, an overall structure of move groups that perform well is found. This structure is then used to describe several potential applications that would effectively use this structure.

In this paper, first a quick overview of the UCB algorithm is presented in Section 2. Then, the research questions to be answered are presented in Section 3. Next, the experimental framework is described in Section 4. Section 5 contains the experiments and their results. Lastly, Section 6 describes potential applications for move groups.

2 The Upper Confidence Bounds Algorithm

The Upper Confidence Bounds (UCB) algorithm is the basis of the UCT algorithm. UCB was designed for a finite-time policy for the multi-armed bandit problem [2]. At each time step, select the machine j that maximizes the following formula:

$$\text{value}_j = \begin{cases} \bar{x}_j + C\sqrt{\frac{\ln(n)}{n_j}} & \text{if } n_j > 0 \\ FPU & \text{if } n_j = 0 \end{cases}$$

Here, $\bar{x}_j$ is the average reward from machine j, n_j is the number of times j has been played, and n is the total number of plays. C is the bias constant which

is $\sqrt{2}$ by default. The FPU value is determined by the first-play urgency policy and is ∞ by default.

This algorithm has been proven to achieve logarithmic regret with any number of playouts. This was then extended to work with trees in the paper that describes the UCT algorithm [6]. Manipulating the bias term affects exploration; a higher constant means more exploration. When first encountering unexplored nodes, the default policy is to explore all children once; alternatively, a customized variation of first-play urgency is implemented. After the search is terminated, the machine or move with the most simulations is selected.

In this paper, UCB parameterized with C, the bias constant, and FPU, the first-play urgency, is the competitor. However, since the addition of move groups creates another level, UCB is used at both levels, which is essentially UCT with a fixed tree.

3 Research Questions

Previous efforts exploring the efficacy of move groups [4] [9] have tried a small subset of possible move groups. This paper examines the research question (RQ) of the perfect move group. RQ1: How do we arrive at the perfect move group that performs better than all other possible move groups? In order to do this, all possible move groups need to be tested and compared to one another. Of course, the perfect move group will be defined by the underlying payoffs of the children. Therefore, RQ2 is to find whether there is a common structure among good move groups so that this might be applied to other problems. The third question (RQ3) is regarding the efficacy of completely random move groups. RQ3: If we know absolutely nothing about the underlying nodes, will a random move grouping perform better than none at all?

4 Experimental Framework

Given the questions to be answered, an artificial game called the 'Multi-armed Bandit Game with Move Groups' was created.

1. The entire game tree is the root node with N children.
2. Each child has a fixed probability of paying off.
3. Using UCB on this tree competes with other trees where move groups have been introduced; an extra level is added into the tree where first a move group is selected by UCB, then a child is selected UCB. No child is duplicated.

Simplicity was required here due to the nature of the questions. Given that structure and behavior of the move groups needed to be investigated, the simplicity allowed for larger experiments to be run to gather information in detail. Especially when N is small, many different move groups can be analyzed.

The policy for selecting the next node to simulate was the modified UCB mentioned before. The bias constant was varied during the experiments with

the following values used for C: 0.0, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0, and 100.0. The policy for first-play urgency used was to have a set value for unexplored nodes. In the default case, FPU was set to 0.5; some of the experiments tested the effects of modifying this value.

5 Experiments and Results

Two quality criteria were used: (1) probability that the best node was selected after a set number of simulations and (2) total regret over time. All experiments were run 5000 times with average results shown.

5.1 Baseline Group Performance

The first experiment exhaustively tried every possible grouping of nine nodes into three groups of three, 280 combinations in total. This allowed for detailed comparison of groups with different payoffs, and values of C and FPU. Each grouping along with standard UCB was run for 8192 simulations. At various intervals, the node that would have been selected was tracked. Performance was measured by the percentage of time that the best node was selected. In practice, knowing the exact value or ranking of nodes is unrealistic, but this experiment provides a baseline to study the potential of groups.

For the first study, the payoff probabilities of the nine nodes were 0.1, 0.2, ..., 0.9. For comparison, results of UCB without move groups are given in Table 1. A 95% confidence interval for the previous selection rates is $\pm 0.71\%$ for 50% and $\pm 1.22\%$ for 75%. The best performing values of C for basic UCB were 0.5 and 1.0 with 0.5 performing slightly better than 1.0 at lower simulation counts and slightly worse at higher simulation counts. To look at the effect of introducing move groups without changing any parameters, the performance of basic UCB was compared to all move groups with the same parameters. The results are shown in Table 2. When the bias term was near its optimal value for basic UCB, introducing move groups almost always decreased the performance. So, in order to find out how to use move groups to increase performance, comparisons must be made with varying C.

Comparing all group and C value pairs gives a good insight into the potential of move groups. When looking at the 32 simulations mark, there are 181 group/C value pairs of a total 2520 (280 groups, 9 values for C) that performed better than the best basic UCB. The most common value for C was 0.5 with a few 1.0 values being present as well. When looking at the 512 simulations mark, there are 224 group/C value pairs that performed better than the best basic UCB. Counting only the number of unique groups, there were 136. For values of C, 1.0, 2.0, 5.0, and 10.0 were all present, with 5.0 performing the best. The two most common structures of groups that performed better than basic UCB at both simulation marks were those that had the best child in a group with the 2nd or 3rd best child or the three best children split among the three groups. The best performing groups had the second structure more often

Table 1. Basic UCB performance based on percentage of the trials when the best arm was selected at various simulation counts with children payoffs ranging from 0.1 to 0.9.

Bias Term	16	32	64	128	256	512	1024	2048	4096	8192
0.0	0.327	0.372	0.396	0.407	0.412	0.414	0.414	0.416	0.416	0.416
0.1	0.354	0.441	0.507	0.550	0.576	0.597	0.613	0.616	0.617	0.621
0.2	0.371	0.497	0.633	0.760	0.817	0.844	0.863	0.871	0.873	0.875
0.5	0.413	0.542	0.733	0.880	0.950	0.982	0.996	0.998	0.998	0.999
1.0	0.214	0.379	0.630	0.832	0.957	0.996	1.000	1.000	1.000	1.000
2.0	0.098	0.333	0.546	0.759	0.914	0.986	0.999	1.000	1.000	1.000
5.0	0.000	0.003	0.387	0.628	0.856	0.956	0.997	1.000	1.000	1.000
10.0	0.000	0.000	0.076	0.395	0.767	0.925	0.993	0.999	1.000	1.000
100.0	0.000	0.000	0.000	0.000	0.000	0.040	0.676	0.991	1.000	1.000

Table 2. Percentage of all groups that performed equally or better than basic UCB with the same parameters. An asterisk marks where basic UCB was selecting the best node 100% of the time.

Bias Term	16	32	64	128	256	512	1024	2048	4096	8192
0.0	12.15	1.43	0.00	0.00	0.36	0.72	1.08	1.08	1.79	1.79
0.1	6.79	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.2	11.08	1.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.5	50.36	45.36	8.22	2.51	3.58	2.86	2.86	2.51	2.51	3.58
1.0	100.00	98.58	86.08	69.29	51.79	25.72	25.72*	50.72*	73.93*	90.36*
2.0	100.00	87.86	68.58	63.22	61.79	67.15	73.22	99.65*	100.00*	100.00*
5.0	100.00	100.00	70.72	61.08	55.36	56.08	57.51	65.36*	98.58*	100.00*
10.0	100.00	100.00	96.08	71.79	56.08	53.22	51.79	56.08	65.01*	81.79*
100.0	100.00	100.00	44.29	100.00	100.00	85.01	64.29	33.93	52.51*	57.15*

than the first. For example, the grouping {{0.1, 0.2, 0.8} {0.3, 0.4, 0.7} {0.5, 0.6, 0.9}} selected the best node 62.24% of the time at 32 simulations compared to 54.22% for basic UCB and 100% of the time at 512 simulations compared to 99.66% for basic UCB.

5.2 Groups Study in Difficult Move Selection

The second iteration of this experiment was run with the exact same parameters except the best child had a payoff probability of 0.81 rather than 0.9. This was to make the problem harder and decrease the occurrence of 100% success rates seen previously. Basic UCB's performance is shown in Table 3. Again, the best observed bias term for basic UCB was around 0.5 and 1.0. When comparing move groups to basic UCB without changing parameters, the results are quite similar to those found for the previous experiment shown in Table 2.

This time, when comparing all group and C value pairs together, the results are more in favor of the groups. Using 32 simulations as the comparison point, 248 pairs and 162 unique groups performed better than the best basic UCB.

The best performing C value of those groups was 0.5, with 0.1, 0.2, and 1.0 present as well. In this case, the top groups have a very obvious structure in common: the best three children are split among the three groups. Using 512 simulations as the next comparison point, 435 pairs and 174 unique groups performed better than the best basic UCB. Interestingly, the same groups as with 32 simulations are present at the top, but with different bias terms. This time, 5.0 and 10.0 are the most common C values and perform the best. At 8192 simulations, the same groups are still present at the top, with a majority having bias terms of 10.0. The group described in the last section is again among the top performers.

Table 3. Basic UCB performance based on percentage of the trials when the best arm was selected at various simulation counts with children payoffs ranging from 0.1 to 0.81.

Bias Term	16	32	64	128	256	512	1024	2048	4096	8192
0.0	0.273	0.306	0.320	0.329	0.333	0.335	0.336	0.336	0.338	0.338
0.1	0.274	0.332	0.374	0.400	0.424	0.444	0.456	0.460	0.461	0.466
0.2	0.286	0.358	0.419	0.455	0.478	0.498	0.515	0.532	0.551	0.570
0.5	0.278	0.335	0.436	0.501	0.551	0.593	0.645	0.690	0.754	0.842
1.0	0.165	0.248	0.360	0.463	0.545	0.595	0.642	0.707	0.779	0.864
2.0	0.079	0.216	0.316	0.411	0.484	0.542	0.611	0.667	0.749	0.842
5.0	0.000	0.002	0.200	0.332	0.432	0.515	0.555	0.617	0.696	0.801
10.0	0.000	0.000	0.067	0.202	0.375	0.478	0.550	0.607	0.675	0.757
100.0	0.000	0.000	0.000	0.000	0.000	0.002	0.268	0.415	0.551	0.667

From these two experiments, we start to see a pattern emerge. In both experiments, groups that did better than basic UCB were those that had the best three children split among the groups. Then, for low simulation counts, a C value of 0.5, near-optimal for basic UCB, performed best. For higher simulation counts, increasing the value by a few orders of magnitude had the best results. This is to counter for the increased selectivity that results from grouping.

Running the same experiment but with payoff probabilities of $0.3, 0.35, ..., 0.7$ had an overall performance between the other two experiments and therefore similar conclusions apply. Changing FPU to 0.3 from 0.5 decreased the performance of all groups and basic UCB. Changing it again to 0.7 from 0.5 had little effect with the largest deviation being 3%.

5.3 Grouping in Games with Large Branching Factor

In order to check whether the best performing groups would have a similar structure with a larger number of children, the experiment was slightly modified for 60 nodes and 10 move groups of 6 nodes each. Values for the nodes were randomly generated and ranged from 0.7% to 92.2%. The second best node had a value of 88.4%. The number of simulations was scaled up to 65536. Since there would be too many groups to test them exhaustively, 20 random groups as well

as two pre-defined groups were generated. The first pre-defined group contained the nodes "in order," that is, the best 6 children in one group, the next best 6 children in another group, and so on. The second pre-defined group was the "distributed" group where the best 10 nodes were split among the groups.

Results of the most experiments were similar to the 9-node experiments. A change from the previous results was the set of C values that performed the best. For basic UCB, 0.2 performed best at simulations counts less than 1024, and 0.5 performed best at higher simulations counts. The best-performing C values for groups also decreased; 0.2 and 0.5 were most common at low simulation counts, and 1.0 and 2.0 were most common at high simulation counts.

The move group structure of the best groups clearly differed from what was expected. Two of the best-performing groups had the best child grouped with the 4th best child. This suggests that the move groups work best when distinguishing between a few good arms rather than many. The fact that the distributed group performed worse than the basic UCB in all but a few cases reinforces this. These scaling results require further study.

5.4 Measuring Group Performance with Regret

Since the UCB algorithm was designed to minimize regret, it was important to look at how the regret of different move groups performs in comparison. Regret for one simulation was calculated by taking the difference of the best node and the chosen node's payoff probabilities. This amount was tracked at every time step and totalled. The values shown are the average total regret over the 5000 trials.

When the payoff probabilities were 0.1, 0.2, ..., 0.9 as in the first experiment, the range of total regret at simulation counts 512 and 8192 for all groups and basic UCB is shown in Figure 1 and 2, respectively. It is interesting to note that the relative shape between the bias terms remains the same no matter the simulation count. A bias term of 0.5 has the least regret throughout the simulations. However, this does not correspond to selecting the best node more often. For example, once the simulation count reaches 1024, some groups select the best node 100% of the time, but only if their bias term is 5.0 or 10.0. This is despite the fact that the regret of those bias terms is very high relative to that of 0.5. Higher bias implies that more time is spent simulating lower value nodes and therefore accumulates more regret. However, this allows the algorithm to distinguish the best node in a better way.

Changing the largest payoff probability to 0.81 yields similar results. The range of total regret at a simulation count of 512 for all groups and basic UCB is shown in Figure 3. Similar results were also found for the 60 node experiment with slightly lower C values having better regret.

From these results, cumulative regret is not the ideal metric for measuring performance of move groups. Even while relative total regret remained the same between simulation counts, a group's preference of C value changed. Since total regret considers all simulations performed rather than the end result, it is biased towards configurations that find good moves quickly and do not deviate. In game

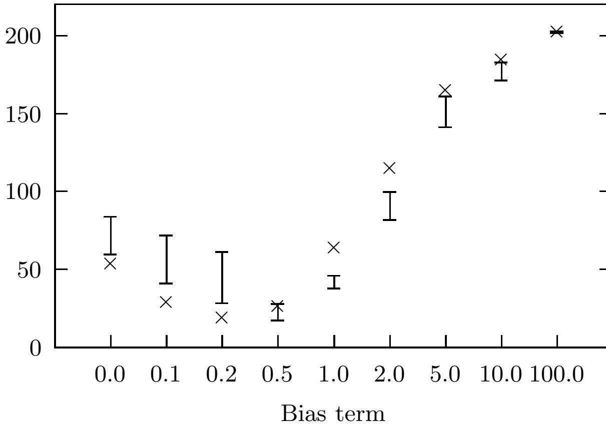


Fig. 1. Total regret of groups, range denoted by bars, and basic UCB, denoted by an x, after 512 simulations with payoff probabilities ranging from 10 to 90%. Data is separated according to the C value.

tree searches, we are more concerned with simple regret: finding the best move regardless of where simulation time was spent. This is similar to work done by Audibert et al. where they endeavor to develop an algorithm that can deal with a situation where the end result is all that matters while still keeping exponentially decreasing regret [1].

5.5 Introducing Move Groups and Using Prior Knowledge

The previous experiments used predetermined groups in order to determine the group structures that performed well given the relative values of the nodes. One possible way to use these structures without knowing the exact value of the underlying nodes is to introduce move groups after a number of simulations. The values used for the nine nodes again was 10-81%. Basic UCB with a bias term of 0.5 was the competitor. Groups were created with various conditions: create groups once after 64, 128, or 512 simulations, create and recreate groups every 50, 100, or 200 simulations, create and recreate groups after each performance measurement. It is interesting to note that the run times of all grouping methods after 8192 simulations were approximately 25% faster than basic UCB due to the fact that fewer nodes are evaluated with groups.

First, creating the best group/bias pair was tried where the top three arms were split among groups and the bias term scaled to higher values with more simulations. Overall, the results were poor with all methods performing 7-12% worse than basic UCB at 8192 simulations which selected the best node 86.5% of the time. Next, the grouping where the best three nodes are grouped together with a bias term of 0.5 was tested. In this case, the performance of creating groups once after 512 simulations was within error bounds of basic UCB. Other

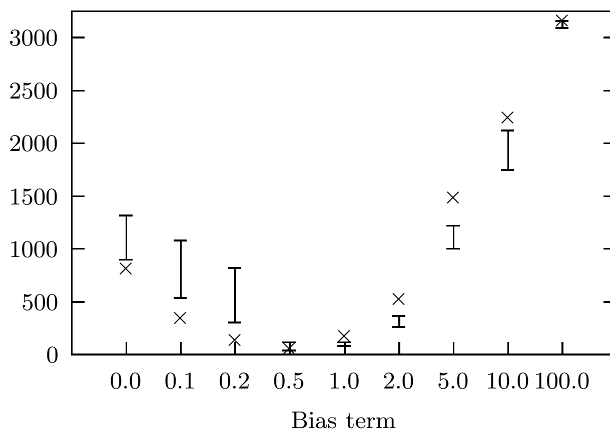


Fig. 2. Total regret of groups, range denoted by bars, and basic UCB, denoted by an x, after 8192 simulations with payoff probabilities ranging from 10 to 90%. Data is separated according to the C value.

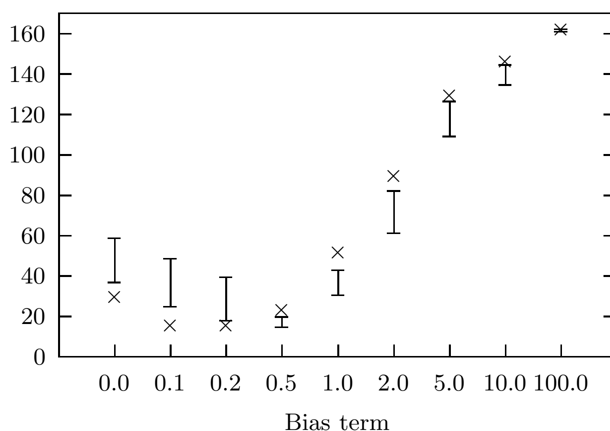


Fig. 3. Total regret of groups, range denoted by bars, and basic UCB, denoted by an x, after 512 simulations with payoff probabilities ranging from 10 to 81%. Data is separated according to the C value.

methods did not perform as well with performance hits of 6-10% compared to basic UCB at 8192 simulations.

To test the effect of prior knowledge, groups were created initially given that the best three nodes are known. The bias term was varied from 0.1 to 10.0. Two grouping methods were tested – separating the best three nodes and grouping them together.

When separating the best nodes, the best performing bias terms were 2.0, 5.0, and 10.0. Initially, they performed worse than basic UCB. After 64 simulations,

they started performing significantly better. However, 2.0 and 5.0 often switched to selecting the second best node rather than the best node at higher simulations. The group using a bias term of 10.0 performed significantly better than basic UCB after 64 simulations and reached 99.8% at 2048 simulations. Putting the best nodes in the same group had more consistent performance, but never better than basic UCB. Bias terms of 0.5, 1.0, and 2.0 all performed within the error bounds of basic UCB.

6 Conclusions and Potential Applications

There are several observations that can be seen in the experimental results. The first is that only introducing arbitrary or random move groups into the search speeds up the search and does not increase the simulation efficiency. This is compounded when there are additional savings in move generation as shown by Lorentz's Amazons program [7]. Second, with very high bias terms, groups on average performed better than basic UCB in the experiments as seen in Table 2.

Third, since random move groups cannot always be used, we need some way to differentiate between the nodes so that they can be grouped properly. The experiments showed that splitting up the best nodes into a few groups had good results regardless of their underlying payoff. Fourth, this was tested using simulation values and prior knowledge. Without prior knowledge, the best results were when the top three nodes were grouped together after a large number of simulations. They performed as well as basic UCB in terms of number of simulations with a 25% increase in speed. Given that only nine nodes is a tough case and the branching factor would typically be decreased much more, this speed increase is expected to persist even with higher node counts. Fifth, with prior knowledge, grouping the best nodes together at the start performed as well as basic UCB and had the speed increase. Sixth, splitting up the best nodes performed better with more simulations and higher bias terms.

From the six observations we may draw two conclusions. First, our research showed that it is possible to define a general structure of good move groups. Second, experiments make it possible to identify the major values of the parameters for enhancing the playing strength.

To exploit the framework in other games, the introduction of move groups during simulations could be used in general. That may prove to be challenging to implement due to the fact that the tree is changed during simulations. If prior knowledge is available, such as using a pattern database to rank moves, then groups could be created based on that ranking during node expansion.

The results described in this paper allow for several interesting avenues of future research, such as introducing move groups during simulations for any game, using pattern databases in Go to rank moves and create groups, and using a heuristic function in chess to rank moves and create groups. Applying approaches to such games remains a topic for future work.

References

1. Audibert, J.-Y., Bubeck, S., Munos, R.: Best Arm Identification in Multi-Armed Bandits. In: COLT 2010, pp. 41–53 (2010)
2. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time Analysis of the Multiarmed Bandit Problem. In: Machine Learning, pp. 235–256 (2002)
3. Chaslot, G.M.J.-B., Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., Bouzy, B.: Progressive Strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation (NMNC) 4, 343–357 (2008)
4. Childs, B.E., Brodeur, J.H., Kocsis, L.: Transpositions and Move Groups in Monte Carlo Tree Search. In: Hingston, P., Barone, L. (eds.) IEEE Symposium on Computational Intelligence and Games, pp. 389–395. IEEE (December 2008)
5. Gelly, S., Silver, D.: Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. Artificial Intelligence 175, 1856–1875 (2011)
6. 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)
7. 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)
8. Müller, M.: Fuego at the Computer Olympiad in Pamplona 2009: a Tournament Report. Technical report, University of Alberta, Dept. of Computing Science, TR09-09 (May 2009)
9. Saito, J.-T., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: Grouping Nodes for Monte-Carlo Tree Search. In: Computer Games Workshop 2007, pp. 276–283 (2007)
10. Wang, Y., Gelly, S.: Modifications of UCT and Sequence-Like Simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, pp. 175–182. IEEE (April 2007)

PACHI: State of the Art Open Source Go Program*

Petr Baudiš and Jean-loup Gailly

Faculty of Mathematics and Physics,
Charles University Prague
pasky@ucw.cz, jloup@gailly.net

Abstract. We present a state of the art implementation of the Monte Carlo Tree Search algorithm for the game of Go. Our PACHI software is currently one of the strongest open source Go programs, competing at the top level with other programs and playing evenly against advanced human players. We describe our implementation and choice of published algorithms as well as three notable original improvements: (1) an adaptive time control algorithm, (2) dynamic komi, and (3) the usage of the criticality statistic. We also present new methods to achieve efficient scaling both in terms of multiple threads and multiple machines in a cluster.

1 Introduction

The board game of Go has proven to be an exciting challenge in the field of Artificial Intelligence. Programs based on the Monte Carlo Tree Search (MCTS) algorithm and the RAVE variant in particular have enjoyed great success in the recent years. In this paper, we present our Computer Go framework PACHI with the focus on its RAVE engine that comes with a particular mix of popular heuristics and several original improvements.

In section 2, we briefly describe the PACHI software. In section 3, we detail the MCTS algorithm and the implementation and heuristics used in PACHI. Section 4 contains a summary of our original extensions to the MCTS algorithm — an adaptive time control algorithm (Sec. 4.1), the dynamic komi method (Sec. 4.2) and our usage of the criticality statistic (Sec. 4.3). In section 5, we detail our scaling improvements, especially the strategy behind our distributed game tree search. Then section 6 summarizes PACHI's performance in the context of the whole Computer Go field.

1.1 Experimental Setup

We use several test scenarios for the presented results with varying number of simulations per move. Often, results are measured only with much faster time settings than that used in real games — by showing different relative contributions of various heuristics, we demonstrate that the aspect of total time available may matter significantly for parameter tuning.

* Supported by the GAUK Project 66010 of Charles University Prague.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 24–38, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In our “low-end” time settings, we play games against GNU Go level 10 [4] with single threaded PACHI, 500 seconds per game, i.e., about 5,000 playouts per move. In the “mid-end” time settings, we play games against FUEGO 1.1 [9] with four PACHI threads, fixed 20,000 playouts per move. In the “high-end” time settings, we play against FUEGO 1.1 as well but using 16 threads, fixed 250,000 playouts per move.

The number of playouts (250,000 for PACHI and 550,000 for FUEGO[1] in “high-end”) was selected so that both use approximately the same time, about 4 seconds/move on average. We use the 19×19 board size unless noted otherwise. In the “high-end” configuration PACHI is 3 stones stronger than FUEGO so we had to impose a large negative komi −50.5 with PACHI always taking white. However, while PACHI scales much better than FUEGO, in the “mid-end” configuration FUEGO and PACHI are about even.[2] The “low-end” PACHI is stronger than GNU GO, therefore PACHI takes white and games are played with no komi. Each test was measured using 5000 games, except for the “low-end” comparisons; we used a different platform and had to take smaller samples.

2 The Pachi Framework

The design goals of PACHI have been (1) simplicity, (2) minimum necessary level of abstraction, (3) modularity and clarity of programming interfaces, and (4) focus on maximum playing strength.

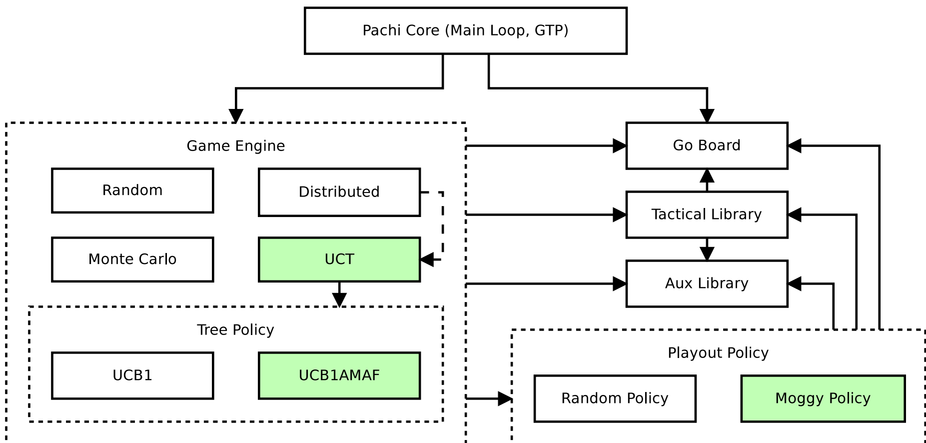


Fig. 1. Block schema of the PACHI architecture. When multiple options of the same class are available, the default module used is highlighted.

[1] The interpretation of *max_games* changed in FUEGO 1.0 such that it includes the count of simulations from reused trees. PACHI does not include them.

[2] When we tried to match FUEGO against the “low-end” PACHI, FUEGO was 110 Elo stronger.

PACHI is free software licenced under the GNU General Public Licence [11]. The code of PACHI is about 17000 lines of pure C, most of the code is richly commented and follows a clean coding style. PACHI features a modular architecture (see Fig. 1): the move selection policy,[3] simulation policy, and other parts reside in pluggable modules and share libraries providing common facilities and Go tools. Further technical details on PACHI may be found in [2].

3 Monte Carlo Tree Search

To evaluate moves, PACHI uses a variant of the Monte Carlo Tree Search (MCTS) — an algorithm based on an incrementally built probabilistic minimax tree. We repeatedly descend the game tree, run a Monte Carlo simulation when reaching the leaf, propagate the result (as a boolean value)[4] back to the root and expand the tree leaf when it has been reached $n = 8$ times.

Algorithm 1. NODEVALUE

Require: $\mathrm{sims}, \mathrm{sims}_{\mathrm{AMAF}}$ are numbers of simulations pertaining the node.
Require: $\mathrm{wins}, \mathrm{wins}_{\mathrm{AMAF}}$ are numbers of won simulations.

$\mathrm{NormalTerm} \leftarrow \frac{\mathrm{wins}}{\mathrm{sims}}$

$\mathrm{RAVETerm} \leftarrow \frac{\mathrm{wins}_{\mathrm{RAVE}}}{\mathrm{sims}_{\mathrm{RAVE}}} = \frac{\mathrm{wins}_{\mathrm{AMAF}}}{\mathrm{sims}_{\mathrm{AMAF}}}$

$\beta \leftarrow \frac{\mathrm{sims}_{\mathrm{RAVE}}}{\mathrm{sims}_{\mathrm{RAVE}} + \mathrm{sims} + \mathrm{sims}_{\mathrm{RAVE}} \cdot \mathrm{sims}/3000}$

$\mathrm{NodeValue} \leftarrow (1 - \beta) \cdot \mathrm{NormalTerm} + \beta \cdot \mathrm{RAVETerm}$

The MCTS variants differ in the choice of the next node during the descent. PACHI uses the RAVE algorithm [5] that takes into account not only per-child winrate statistics for the move being played *next* during the descent, but also (as a separate value) *anytime later*[5] during the simulation (the so-called AMAF statistics). Therefore, we choose the node with the highest value given by Algorithm 1 above, a simplified version of the RAVE formula [5] (see also Sec. 4.2).

3.1 Prior Values

When a node is expanded, child nodes for all the possible followup moves are created and pre-initialized in order to kick-start exploration within the node. The value (expectation) for each new node is seeded as 0.5 with the weight of several virtual simulations, we have observed this to be important for RAVE stability. The value is further adjusted by various heuristics, each contributing ε fixed-result virtual simulations ("equivalent experience" $\varepsilon = 20$ on 19×19, $\varepsilon = 14$ on 9×9). (This is similar to the progressive bias [7], but not equivalent.)

[3] The default policy is called "UCT", but this is only a traditional name; the UCT exploration term is not used by default anymore.

[4] In the past, we have been incorporating the final score in the propagated value, however this has been superseded by the Linear Adaptive Komi (Sec. 4.2).

[5] Simulated moves played closer to the node are given higher weight as in FUEGO [9].

Table 1. Elo performance of various prior value heuristics on 19×19

Heuristic	Low-end	Mid-end	High-end
w/o eye malus	-31 ± 32	-3 ± 16	$+1 \pm 16$
w/o ko prior	-15 ± 32	-3 ± 16	$+10 \pm 16$
w/o 19×19 lines	-15 ± 33	-4 ± 16	-6 ± 16
w/o CFG distance	-66 ± 32	-66 ± 16	-121 ± 16
w/o playout policy	-234 ± 42	-196 ± 16	-228 ± 16

Most heuristics we use and the mechanism of equivalent experience are similar to the original paper on MOGO [5]. Relative performance of the heuristics is shown in Table 1. The Elo difference denotes the change of the program strength when the heuristic is disabled. It is apparent that the number of simulations performed is important for evaluating heuristics. The following six heuristics are applied.

- The "eye" heuristic adds virtual lost simulations to all moves that fill single-point true eyes of our own groups. Such moves are generally useless and not worth considering at all; the only exception we are aware of is the completion of the "bulky five" shape by filling a corner eye, this situation is rare but possible, thus we only discourage the move using prior values instead of pruning it completely.
- We encourage the evaluation of ko fights by adding virtual wins to a move that re-takes a ko no more than 10 moves old.
- We encourage sane 19×19 play in the opening by giving a malus to first-line moves and bonus to third-line moves if no stones are in the vicinity.

Table 2. The ε values for the CFG heuristic

	$\delta = 1$	$\delta = 2$	$\delta = 3$
19×19	55	50	15
9×9	45	40	15

- We encourage the exploration of local sequences by giving bonus to moves that are close to the last move based on δ, the length of the shortest path in the Common Fate Graph [13], with variable ε set as shown in Fig. 2. This has two motivations — first, with multiple interesting sequences available on the board, we want to ensure the tree does not swap between situations randomly but instead reads each sequence properly. Second, this is well rooted in the traditional Go strategy where a large portion of moves is indeed "sente", requiring a local reply.
- We give joseki moves twice the default ε (using the joseki dictionary described below). This currently has no performance impact.

Table 3. Elo performance of some playout heuristics on 19×19

Heuristic	Low-end	Mid-end	High-end
w/o Capture	-563 ± 106	-700	-949
w/o 2-lib.	-86 ± 34	—	—
w/o 3×3 pats.	-324 ± 37	-447 ± 34	-502 ± 36
w/o self-atari	-34 ± 33	—	-31 ± 16

– We give additional priors according to the suggestions by the playout policy. The default ε is halved for multi-liberty group attack moves.

3.2 Playouts

In the game simulations (playouts) started at the leaves of the Monte Carlo Tree, we semi-randomly pick moves until the board is completely filled (up to one-point eyes). The move selection should be randomized, but heuristics allowing for realistic resolution of situations in various board partitions are highly beneficial.

We use the MoGo-like rule-based policy [12] that puts emphasis on localized sequences and matching of 3×3 "shape" board patterns. Heuristics are tried in a fixed order and each is applied with certain probability p, by default $p = 0.8$ for 19×19 and $p = 0.9$ for 9×9.[6] A heuristic returns a set of suggested moves; if the set is non-empty, a random move from the set is picked and played, if the set is empty (the common case), the next heuristic is tried. If no heuristic matches, a uniformly random[7] move is chosen.

See Table 3 for relative performance of the heuristics with the largest impact[8] (the Elo difference again denotes strength change when the heuristic is disabled).

We apply the following three main rules.

– With $p = 0.2$, ko is re-captured if the opponent played a ko in the last 4 moves.
– Local checks are performed — heuristics applied in the vicinity of the last move.
  • With $p = 0.2$, we check if the liberties of the last move group form a "nakade" shape.[9]
  • If the last move has put its own group in atari, we *capture* it with $p = 0.9$. If it has put a group of ours in atari, we attempt to escape or counter-capture other neighboring groups.

[6] Some of the precise values below are 19×19 only, but that is mainly due to a lack of tuning for 9×9 on our part.
[7] Up to one-point eye filling and the self-atari filter described later.
[8] Some of the low-probability heuristics represent only a few Elo of improvement and could not have been re-measured precisely with the current version.
[9] I.e., if we could kill the group by playing in the middle of the group eyespace.

- If the last move has reduced its own group to just *two liberties*, we put it in atari, trying to prefer atari with low probability of escape; if the opponent has reduced our group to two liberties, we attempt either to escape or to put some neighboring group in atari, aiming to handle the most straightforward capturing races.
- With $p = 0.2$, we attempt to do a simplified version of the above (safely escaping or reducing liberties of a neighboring group) for groups of three and four liberties.
- Points neighboring the last two moves are (with $p = 1$) matched for 3×3 board patterns centered at these points similar to patterns presented in [12], extended with information on "in atari" status of stones. We have made a few minor empirical changes to the pattern set.

– We attempt to play a joseki[10] followup based on a board quadrant match in a hash table. The hash table has been built using the "good variations" branches of the Kogo Joseki Dictionary [17]. This has a non-measurable effect on the performance against other programs, but makes PACHI's play prettier for human opponents in the opening.

The same set of heuristics is also used to assign prior values to new tree nodes (as described above). Bad self-atari moves are pruned from heuristic choices and stochastically also from the final random move suggestions: in the latter case, if the other liberty of a group that is being put in self-atari is safe to play, it is chosen instead, helping to resolve some tactical situations involving shortage of liberties and false eyes.

4 MCTS Extensions

Below we discuss three MCTS extensions: Time Control (in 4.1), Dynamic Komi (in 4.2), and Criticality (in 4.3).

4.1 Time Control

We have developed a flexible time allocation strategy when the total thinking time is limited, with the goal of the longest search in the most critical parts of the game — in the middle game and particularly when the best move is unclear.

We assign two time limits (and a fixed delay for network lag and tree management overhead) for the next move — the *desired* time t_d and *maximum* time t_m. Only t_d time is initially spent on the search, but this may be extended up to t_m in case the tree results are too unclear (which triggers very often in practice).

Given the main time T and estimated number of remaining moves in the game[11] R, the default allocation is $t_d = T/R$ and $t_m = 2\,t_d$, recomputed on each move so that we account for any overspending.

[10] Common move sequence, usually played in a corner in the game beginning.

[11] We assume that on average, 25% of board points will remain unoccupied in the final position. We always assume at least 30 more moves will be required.

Furthermore, we tweak this allocation based on the move number so that t_d peaks in the middle game. Let t_M be the maximum time t_m at the end of the middle game (40% of the board has been played). In the beginning, we linearly increase the default t_d up to t_M until 20% of the board has been played (the beginning of the middle game) and set $t_d = t_M$ for the whole middle game. After this, or if we are in byoyomi, the remaining time is spread uniformly as described above.

For overtime (byoyomi), we use our *generalized overtime* specification: after the main time elapses, fixed-length overtime T_o for each next m moves is allocated, with n overtime periods available. Japanese byoyomi is a specific case with $m = 1$ while Canadian byoyomi implies $n = 1$. If overtime is used, the main time is still allocated as usual, except that $t_m = 3\,t_d$; furthermore, the lower time for t_d of the main time is the t_d for byoyomi, and the first $n - 1$ overtime periods are spent as if they were part of the main time. The time per move in the last overtime period is allocated as $t_d = T_o/m$ and $t_m = 1.1\,t_d$.

The search is terminated early if the expectation of win is very high $\mu \geq 0.9$ or the chosen move cannot change[12] anymore. In contrast, the tree search continues even after t_d time already elapsed if the current state of the tree is unclear, i.e., either of the following three conditions triggers.

- The expectations of the best move candidate μ_a and its best reply μ_{aa} are too different, $best_r = |\mu_a - \mu_{aa}| > 0.02$.
- The two best move candidates are equally simulated, i.e., $best_2 = \varepsilon_a/\varepsilon_b < 2.5$ for the playout counts $\varepsilon_a, \varepsilon_b$ of the two best moves.
- The best move (root child chosen based on the most simulations) is not the move with the highest win expectation.

Figures 2 and 3 show that using such a flexible time strategy results in an increase of an up to 80 Elo points performance compared to the baseline.[13]

The recently published time allocation by Erica [14] is similar, but while we focus on over-spending strategies, Erica focuses more on the middle game time

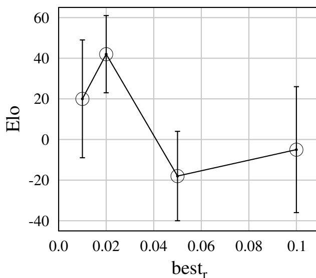


Fig. 2. Best — best child ratio $best_r$

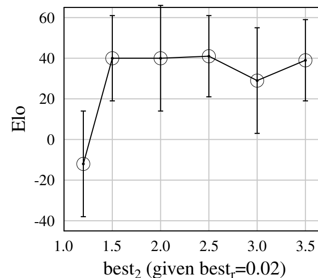


Fig. 3. Best — second best delta $best_2$

[12] We choose the most simulated node as the move to play. We terminate search early if the most simulated node cannot change even if it did not receive any more simulations for the rest of the t_m time.

[13] We used a very fast "low-end" scenario with 300 seconds per game.

allocation (adjusting time non-linearly, for a peak, not a plateau). Our middle game time allocation algorithm is not a source of significant performance benefits and we expect that our and ERICA's algorithm could be reconciled.

4.2 Dynamic Komi

The MCTS algorithm evaluates the possible moves most accurately when the winning rate is near 50%. If most simulations are won or lost, the resolution of the evaluation naturally gets more coarse. Such "extreme situations" are fairly common in Go — especially in handicap games or in the endgame of uneven matches. The dynamic komi technique [1] aims to increase MCTS performance in such situations by shifting the win-loss score threshold (komi) from zero to a different value; if a player winning 90% of simulations is required to win by a margin of 10 points instead of just 1 point, we can expect the winning rate to drop and the game tree will obtain information with a slight bias but also with a much higher resolution.

In PACHI, we have previously successfully employed mainly the Linearly Decreasing Handicap Compensation (LDHC) and Value-based Situational Compensation (VSC) [1]. Recently, we have introduced another novel method: the *Linear Adaptive Komi*. It uses LDHC up to a fixed number of moves, then increases the komi against PACHI if it wins with probability above a fixed sure win threshold (we use 0.85). This retains a good performance of LDHC for handicap games and allows winning by a large point margin when PACHI has a comfortable lead. Without this, PACHI is indifferent for the discrepancies between the winning moves, often steering to a 0.5 point win by playing what humans consider silly moves. At the same time, this strategy is more straightforward and more robust than VSC.

4.3 Criticality

RAVE improves over the plain MCTS by using approximate information on move performance gathered from related previous simulations. We can supply further information using the *point criticality* — the covariance of owning a point and winning the game [8,18],

$$Crit(x) = \mu_{win(x)} - (2\mu_{b(x)}\mu_{\mathrm{b}} - \mu_{b(x)} - \mu_{\mathrm{b}} + 1)$$

with $\mu_{b(x)}$ and $\mu_{w(x)}$ being the expectations of Black and White owning the coordinate, and μ_{b} and μ_{w} the expectations of a player winning the game.

The criticality measure itself has been already proposed in the past. We introduce an effective way to incorporate criticality in the RAVE formula — increasing the proportion of won RAVE simulations in the nodes of critical moves.

$$sims_{RAVE} = (1 + c \cdot Crit(x)) \cdot sims_{AMAF}$$

$$wins_{RAVE} = (1 + c \cdot Crit(x)) \cdot wins_{AMAF}$$

We track criticality in each tree node based on the results of simulations coming through the node. We use criticality only when the node has been visited

at least $n = 2000$ times. $c = 1.1$ yields an improvement of approximately 10 Elo points in the "high-end" scenario, but $n = 192$ can achieve as much as 69 ± 32 Elo points in the "low-end" scenario. More details on this way of criticality integration may be found in [2]. A dynamic way of determining n may make the improvement more pronounced in the future.

5 Parallelization

Pachi supports both shared memory parallelization and cluster parallelization. It is highly scalable with more time or more threads, and scales relatively well with more cluster nodes. Fig. 4 shows the general scaling of Pachi. In all the distributed experiments and most single machine experiments, Fuego and Pachi both use 16 threads per machine and a fixed number of playouts. The measurements are done on a 19×19 board.

5.1 Shared Memory Parallelization

Historically, various thread parallelization approaches for MCTS have been explored [6]. In Pachi, we use the *in-tree parallelization,* with multiple threads performing both the tree search and simulations in parallel on a shared tree and performing lock-free tree updates [9]. To allocate all children of a given node, Pachi does not use a per-thread memory pool, but instead a pre-allocated global

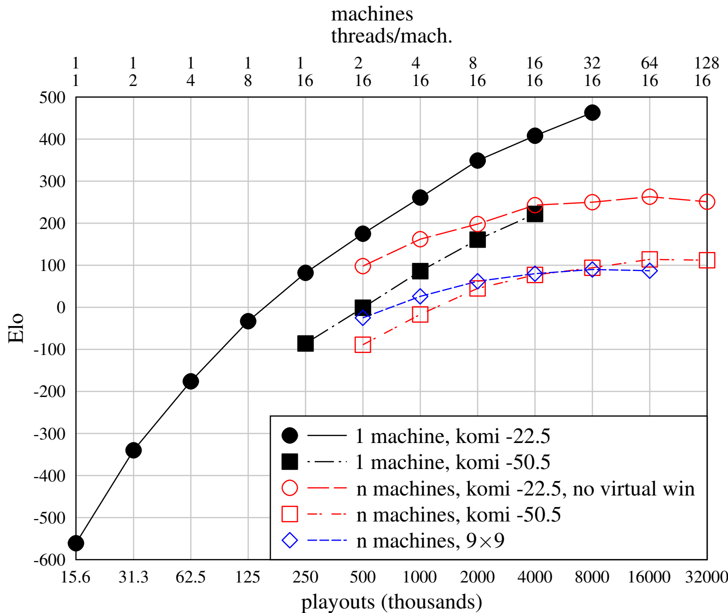


Fig. 4. Thread and distributed scalability

node pool and a single atomic increment instruction updating the pointer to the next free node. The leftmost curve in Fig. 4 shows scaling performance on a single machine when raising the number of playouts per move from 15,625 to 8,000,000 against a constant opponent (FUEGO 1.1 with 550,000 playous per move). To allow an average time per move of at least 2 seconds, the number of threads was reduced for PACHI up to 125,000 playouts per move; above this, the number of threads was kept constant at 16 and so the time per move increased (up to 2 minutes per move for 8,000,000 playouts).

The results show that PACHI is extremely scalable on a single machine. The strength improvement is about 100 Elo points per doubling in the middle of the range where PACHI and FUEGO have equal resources (16 threads each and same time per move). The improvement drops to about 50 Elo points per doubling at the high end, but shows no sign of a plateau[14]. Segal [20] reports a similar scalability curve, but with measurements limited to self-play and 4 hours per game. To measure the effect of the opponent strength, experiments were performed with both komi −22.5 and −50.5. As seen in Fig. 4, the curves are quite similar in both cases, only offset by a roughly constant number of Elo points.

To connect the concept of negative komi and handicap stones, we measured the Elo variation with a variable handicap and komi, as shown in Figs. 5 and 6. Against FUEGO, one extra handicap stone is measured to be worth approximately 70 Elo points and 12 points on the board. (The effect of one handicap stone would be larger in self-play.)

Most experiments were done with a constant number of playouts to improve the accuracy of the Elo estimates. We also measured the performance with fixed total time (15 minutes per game plus 3 seconds/move byoyomi) and a variable number of cores for PACHI.[15] The timed experiments in Fig. 7 demonstrate excellent scalability up to 22 cores[16].

Figures 7 and 8 show inflation of self-play experiments compared to games against a different reference opponent. Scalability results are most often reported only in self-play. This is in our opinion rather misleading. Self-play scalability is far easier than scalability against an opponent that uses different algorithms, for example +380 Elo points instead of +150 Elo points when doubling from 1 to 2 cores. The same effect is also visible in the distributed mode as shown in Fig. 10, where the 128-machine version is 340 Elo points stronger than the 2-machine version in self-play but only 200 Elo points stronger against FUEGO. Given the availability of at least two strong open-source Go programs (PACHI and FUEGO), we strongly encourage other teams to report scalability results against other opponents rather than self-play.

[14] We could not go beyond 8,000,000 playouts/move because of the resource requirements for 5000 games at more than 9 hours per game each.

[15] Since FUEGO lost on time far too frequently even with byoyomi, only PACHI used fixed time and FUEGO used a constant number of playouts.

[16] The timed experiments were run on 24-cores machines, with at least 2 cores reserved for other processes.

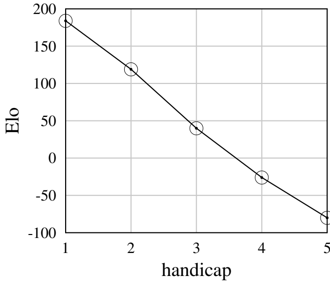


Fig. 5. Strength variation with handicap

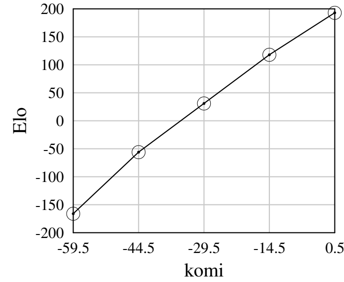


Fig. 6. Strength variation with komi

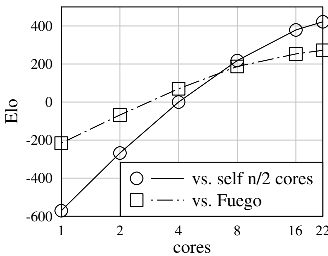


Fig. 7. Self-play scalability (fixed time)

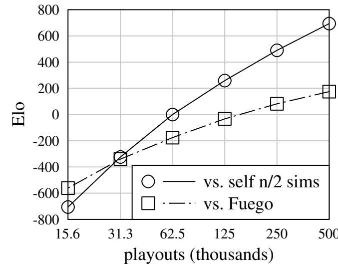


Fig. 8. Self-play scalability (fixed sims)

Fig. 9 shows the strength speedup relative to the number of cores, i.e., the increase in playing time needed to achieve identical strength play [6,20]. The tests are done using the "high-end" scenario, but Pachi uses a variable number of cores. The speedup is perfect (within the error margin) up to 22 cores — for 22 cores the measured speedup is 21.7 ± 0.5.

5.2 Cluster Parallelization

The MCTS cluster parallelization is still far from being clearly solved. Pachi features elaborate support for distributed computations with information exchange between the nodes, but it still scales much slower when multiplying the number of nodes rather than processors with a low-latency shared tree. The cluster version with 64 nodes is about 3 stones stronger than the single machine version. Node statistics are sent using TCP/IP from slave machines to one master machine, merged in the master, and the merged results are sent by the master back to the slaves. Only updates relative to the previously sent results are exchanged, to minimize the network traffic. The network is standard 1 Gb/s Ethernet, so it was critical to optimize it. Statistics are sent only for the first n levels in the tree. Surprisingly, we found the value $n = 1$ to be optimal (i.e., only the information about the immediate move candidates is shared). Understanding this should be the subject of further study. Mogo [3] goes up to $n = 3$ but it uses a

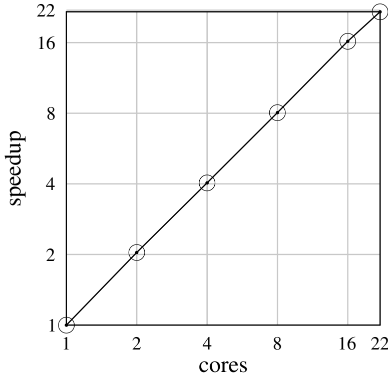


Fig. 9. Strength speedup.

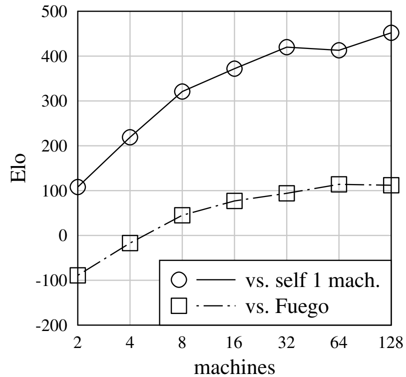


Fig. 10. Self-play scalability in the distributed mode

high performance network (such as Myrinet or InfiniBand) whereas PACHI uses standard Ethernet.

The distributed protocol was designed to be extremely fault tolerant. Nodes can be shut down arbitrarily. The PACHI processes run at the lowest possible priority and can be preempted at any time. The master sums the contributions of all slaves and plays the move most popular among them. In timed games, the master plays when more than half of the slaves indicate that they are willing to play now, or when the time runs out. In experiments with fixed number of playouts, the master plays when half of the slaves have reached their threshold, or when the total number of playouts from all slaves reaches a global threshold. These two tests can trigger quite differently in the presence of flaky slaves. We have used the former method for all the experiments reported here, but the latter method improves scalability further.

Virtual loss [6] aims to spread parallel tree descents — a virtual lost simulation is added to each node visited during the descent and removed again in the update phase. We have found that cluster parallelization is significantly more efficient if multiple lost simulations are added; we use $n = 6$. This encourages different machines to work on different parts of the tree, but increasing exploration by multiple virtual losses slighly improves the single machine case as well.

To encourage further diversity among machines, we introduced the concept of *virtual win*. Each node is given several virtual won simulations in a single slave (if the node number modulo number of slaves equals the slave number), therefore different nodes are encouraged to work on different parts of the tree, this time in a deterministic manner. We use a different number of virtual wins for children of the root node ($n = 30$) and for other nodes ($n = 5$). We also tried to use losses instead of wins; the results were similar so we kept using wins to avoid confusion with the quite different concept of virtual loss. Virtual losses encourage diversity between threads on a single machine; virtual wins encourage diversity between machines.

Fig. 4 shows that virtual wins measurably the improved scalability in a distributed mode. Going from 2 to 64 machines improved the strength by 160 Elo points without virtual win and by 200 Elo points with virtual win (compare the lines for komi −22.5 and komi −50). Distributed Depth-First UCT [23] probably performs better but it is significantly more complex to implement, while multiple virtual loss and virtual win only require a few lines of code.

Fig. 4 also shows that distributed scalability for 9×9 games is harder than for 19×19 games, confirming reports by the MoGo team [21]. The average depth of the principal variation was measured as 28.8 on 9×9 with 4 minutes total time per game, and 14.7 on 19×19 with 29 minutes total time per game.

6 Overall Performance

Pachi's primary venue for open games with the members of the public is the KGS internet Go server [19]. Instances running with 8 threads on Intel i7 920 (hyperthreading enabled) and 6 GiB of RAM can hold a solid 1-dan rank; a cluster of 64 machines with 22 threads each is ranked as 3-dan. (The top program on KGS Zen has the rank of 5-dan.) Distributed Pachi regularly participates in the monthly KGS tournaments [22], usually finishing on the second or third place, but also winning occasionally, e.g., in the August 2011 KGS Bot Tournament.

The cluster Pachi participated in the Human vs. Computer Go Competition at SSCI 2011, winning a 7-handicap 19×19 match against Zhou Junxun 9-dan professional [16]. Zhou Junxun commented that Pachi played on a professional level when killing an invading white group (the bulk of the game). In the European Go Congress 2011 Computer Go tournament [10], distributed Pachi tied with Zen for the first place in the 19×19 section.

In addition to algorithmic improvements, an enormous amount of tuning of over 80 different parameters also significantly improved Pachi's strength. However, at most one out of ten experiments results in a positive gain. Moreover, improvements become harder as the program gets stronger. For example, multiple virtual loss and virtual win initially provided a significant performance boost (30 Elo points each), but after other unrelated algorithmic improvements, their combined effect is now under 10 Elo points per doubling.

For this reason, we have also omitted full graphs of performance based on the values of various constants but describe just the optimal values. While we have originally explored the space of each parameter, resource limitations do not allow us to re-measure the effect of most parameters after each improvement. We can only make sure that we remain in the local optimum in all dimensions.

7 Conclusion

We have described a modern open source[17] Computer Go program Pachi. It features a modular architecture, a small and lean codebase, and a top-performing

[17] The program source can be downloaded at http://pachi.or.cz/.

implementation of the Monte Carlo Tree Search with RAVE and many domain-specific heuristics. The program continues to demonstrate its strength by regularly playing on the internet, with both other programs and people.

We have also introduced various extensions of the previously published methods. Our adaptive time control scheme allows PACHI to spend most time on the most crucial moves. Dynamic komi allows the program to cope efficiently with handicap games. A new way to apply the criticality statistic enhances the tree search performance. PACHI scales well thanks to multiple-simulation virtual loss and to our distributed computation algorithm including virtual win.

Acknowledgments. We have borrowed useful implementation tricks and interesting ideas from other open source programs FUEGO, GNU GO and LIBEGO [15]. Jan Hric offered useful comments on early versions of the paper. We would also like to thank the anonymous referees for their helpful suggestions.

References

1. Baudiš, P.: Balancing MCTS by dynamically adjusting komi value. ICGA Journal (in review)
2. Baudiš, P.: Information Sharing in MCTS. Master's thesis, Faculty of Mathematics and Physics, Charles University Prague (2011)
3. Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hoock, J.-B., Hérault, T., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P., Yu, Z.: Scalability and Parallelization of Monte-Carlo Tree Search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 48–58. Springer, Heidelberg (2011)
4. Bump, D., Farneback, G., Bayer, A., et al.: GNU Go, http://www.gnu.org/software/gnugo/gnugo.html
5. Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., Teytaud, O.: Adding Expert Knowledge and Exploration in Monte-Carlo Tree Search. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 1–13. Springer, Heidelberg (2010)
6. 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)
7. Chaslot, G., Winands, M., van den Herik, H.J., Uiterwijk, J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. In: Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session (2007), http://www.math-info.univ-paris5.fr/~bouzy/publications/CWHUB-pMCTS-2007.pdf
8. Coulom, R.: Criticality: a Monte-Carlo heuristic for Go programs. University of Electro-Communications, Tokyo, Japan (2009), Invited talk, http://remi.coulom.free.fr/Criticality/
9. Enzenberger, M., Müller, M., Arneson, B., Segal, R.: Fuego — an open-source framework for board games and Go engine based on Monte-Carlo Tree Search. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 259–270 (2010)
10. European Go Federation: European Go Congress 2011 in Bordeaux, Computer Go (2011), http://egc2011.eu/index.php/en/computer-go

11. Free Software Foundation: GNU General public licence (1991), http://www.gnu.org/licenses/gpl-2.0.html
12. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Research Report RR-6062, INRIA (2006), http://hal.inria.fr/inria-00117266/en/
13. Graepel, T., Goutrié, M., Krüger, M., Herbrich, R.: Learning on Graphs in the Game of Go. In: Dorffner, G., Bischof, H., Hornik, K. (eds.) ICANN 2001. LNCS, vol. 2130, pp. 347–352. Springer, Heidelberg (2001)
14. Huang, S.C., Coulom, R., Lin, S.S.: Time management for monte-carlo tree search applied to the game of go. In: 2010 International Conference on Technologies and Applications of Artificial Intelligence (TAAI), pp. 462–466 (November 2010)
15. Lew, Ł.: libEGO — Library of effective Go routines, https://github.com/lukaszlew/libego
16. National University of Taiwan: Human vs. Computer Go competition. In: SSCI 2011 Symposium Series on Computational Intelligence (2011), http://ssci2011.nutn.edu.tw/result.htm
17. Odom, G., Ay, A., Verstraeten, S., Dinerstein, A.: Kogo's joseki dictionary, http://waterfire.us/joseki.htm
18. Pellegrino, S., Hubbard, A., Galbraith, J., Drake, P., Chen, Y.P.: Localizing search in Monte-Carlo Go using statistical covariance. ICGA Journal 32(3), 154–160 (2009)
19. Schubert, W.: KGS Go Server, http://gokgs.com/
20. Segal, R.: On the Scalability of Parallel UCT. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 36–47. Springer, Heidelberg (2011)
21. Teytaud, O.: Parallel algorithms (2008), http://groups.google.com/group/computer-go-archive/msg/d1d68aaa3114b393
22. Wedd, N.: Computer Go tournaments on KGS (2005-2011), http://www.weddslist.com/kgs/index.html
23. Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H., Ishikawa, Y.: Scalable distributed Monte-Carlo Tree Search. In: Borrajo, D., Likhachev, M., Lopez, C.L. (eds.) SOCS, pp. 180–187. AAAI Press (2011)

Time Management for Monte-Carlo Tree Search in Go

Hendrik Baier and Mark H.M. Winands

Games and AI Group, Department of Knowledge Engineering,
Maastricht University, Maastricht, The Netherlands
{hendrik.baier,m.winands}@maastrichtuniversity.nl

Abstract. The dominant approach for programs playing the game of Go is nowadays Monte-Carlo Tree Search (MCTS). While MCTS allows for fine-grained time control, little has been published on time management for MCTS programs under tournament conditions. This paper investigates the effects that various time-management strategies have on the playing strength in Go. We consider strategies taken from the literature as well as newly proposed and improved ones. We investigate both *semi-dynamic* strategies that decide about time allocation for each search before it is started, and *dynamic* strategies that influence the duration of each move search while it is already running. In our experiments, two domain-independent enhanced strategies, EARLY-C and CLOSE-N, are tested; each of them provides a significant improvement over the state of the art.

1 Introduction

In tournament gameplay, time is a limited resource. *Sudden death*, the simplest form of time control, allocates to each player a fixed time budget for the whole game. If a player exceeds this time budget, he[1] loses the game immediately. Inasmuch as longer thinking times result in stronger moves, the player's task is to distribute his time budget wisely among all moves in the game. This is a challenging task both for human and computer players. Previous research on this topic [1,7,14,19,21] has mainly focused on the framework of $\alpha\beta$ search with iterative deepening. In a number of game domains however, this algorithm is more and more losing its appeal.

After its introduction in 2006, *Monte-Carlo Tree Search* (MCTS) [5,15] has quickly become the dominant paradigm in computer Go [17] and many other games [18]. Unlike for $\alpha\beta$ search, relatively little has been published on *time management* for MCTS [3,13]. MCTS however allows for much more fine-grained time-management strategies due to its *anytime* property. It can be stopped after every playout and return a move choice that makes use of the complete search time so far, while $\alpha\beta$ searchers can only make use of completely explored root moves of a deepening iteration.

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

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 39–51, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper, we systematically test and compare a variety of time-management strategies for MCTS in computer Go. We include newly proposed strategies as well as strategies described in [3] and [13], partly in enhanced form. Experiments in 13×13 and 19×19 Go are described, and a significant improvement of the state of the art is demonstrated.

This paper is organized as follows. Section 2 gives an overview of related work on time management for game-playing programs in general and Go programs in particular. Section 3 outlines the approaches to time management studied in this paper, while Section 4 presents experimental results in Go. Conclusions and future research follow in Section 5.

2 Time Management

The first publication to address the topic of time management in computer games was by Hyatt [14]. He observed that human chess grandmasters do not use an equal amount of time per move, but play standard openings quickly, think longest directly after coming out of the opening, and then play increasingly fast towards the end of the game. He also suggested a technique that lets $\alpha\beta$ search explore a position longer to find a better move if the best move of the last deepening iteration turns out to lose material.

Donninger [7] gave four "golden rules" for the use of time during a chess game, both for human and computer players: "a) Do not waste time in easy positions with only one obvious move. b) Use the opponent's thinking time effectively. c) Spend considerable time before playing a crucial move. d) Try to upset the opponent's timing." He considered rule c) to be the most important one by far, but also the hardest. In this paper, we try to approach rules a) and c) simultaneously by attempting to estimate the difficulty of a position and adjusting search time accordingly.

Althöfer *et al.* [1] published the first systematic evaluation of time-management algorithms for chess. Amongst others, strategies were proposed to identify trivial moves that can be made quickly, as well as troublesome positions that require more thinking. The time controls considered, typical for chess, specify a given amount of time for a given number of moves. They are insofar different from sudden death as used in this paper as it here does not refer to the number of moves by the player, but only to the total amount of time per game.

Markovitch and Sella [19] used the domain of checkers to acquire automatically a simple time-allocation strategy, distributing a fixed number of deep searches among the moves of a game. The authors divided time-management strategies into three categories. (1) *Static* strategies decide about time allocation to all future moves before the start of the game. (2) *Semi-dynamic* strategies determine the computation time for each move before the start of the respective move search. (3) *Dynamic* strategies make "live" timing decisions while the search process is running. This categorization is used in the remainder of this paper.

Šolak and Vučković [21] devised and tested a number of time-management models for modern chess engines. Their model M2a involved the idea of estimating the remaining number of moves, given the number of moves already played,

from a database of master games. We use a similar approach as the basis for our strategies. In more sophisticated models, Šolak and Vučković developed definitions for the complexity of a position—based on the number of legal moves—and allocated time accordingly. Since the number of legal moves is not a suitable measure in the game of Go, we use the concept of criticality [6] instead to identify important positions.

Kocsis *et al.* [16] compared temporal difference learning and genetic algorithms for training a neural network to make semi-dynamic timing decisions in the game Lines of Action. The network could set the underlying $\alpha\beta$ program to one of three predefined search depths.

For the framework of MCTS, only two publications exist so far. Huang *et al.* [13] evaluated a number of time-management heuristics for 19×19 Go, assuming sudden-death time controls. As described in Subsection 4.1, we implemented and optimized their heuristics as a baseline for our approaches. The ideas of the "unstable evaluation" heuristic (UNST) and the "think longer when behind" heuristic (BEHIND) were first described and tested in [13].

During the preparation of this paper, Baudiš [3] published remarks on time management for the state-of-the-art Go program PACHI in his Master's thesis. Ideas similar to our "close second" (CLOSE) and "early exit" heuristics (EARLY) were here formulated independently.

3 Time-Management Strategies

In this section, we describe first the semi-dynamic (3.1), and then investigate the dynamic time-management strategies (3.2).

3.1 Semi-dynamic Strategies

The following five strategies determine the search time for each move directly before the search for this move is started.

EXP. The straightforward EXP strategy for time allocation, used as the basis of all further enhancements in this paper, divides the remaining thinking time for the entire game ($t_{\text{remaining}}$) by the expected number of remaining moves for the player (m_{expected}) and uses the result as the search time for the next move (t_{nextmove}). The formula is as follows:

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{1}$$

m_{expected} can be estimated in various ways. Three heuristics are investigated in this paper, two of them are game-independent and one is game-specific. The first game-independent heuristic (EXP-MOVES) estimates the number of remaining moves given the number of moves already played. The second game-independent heuristic (EXP-SIM) estimates the number of remaining moves given the length of simulated games in the preceding move search.

The third heuristic (EXP-STONES) is specific to the game of Go and uses the number of stones on the board as an estimator of remaining game length. Other games may or may not provide other indicators. The parameters for all three heuristics, e.g., the precise mapping from played moves to remaining moves for EXP-MOVES, are set to their average values in a large set of games played in self-play.

OPEN. The OPEN strategy puts emphasis on the opening phase of the game. Formula 2 modifies the search time for every move in the game by multiplying it with a constant "opening factor" $f_{\text{opening}} > 1$.

$$t_{\text{nextmove}} = f_{\text{opening}} \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \quad (2)$$

This results in more time per move being used in the beginning of the game than at the end. As opposed to the implicit assumption of Formula 1 that equal time resources should be allocated to every expected move, here it is assumed that the first moves of a game have greater influence on the final outcome than the last moves and thus deserve longer search times.

MID. Instead of moves in the opening phase, the MID strategy increases search times for moves in the middle game, which can be argued to have the highest decision complexity of all game phases [13]. For this purpose, the time as given by Formula 1 is increased by a percentage determined by a Gaussian function over the set of move numbers, using three parameters a, b, and c for height, position, and width of the "bell curve".

$$f_{\text{Gaussian}}(x) = ae^{-\frac{(x-b)^2}{2c^2}} \quad (3)$$

$$t_{\text{nextmove}} = (1 + f_{\text{Gaussian}}(\text{current move number})) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \quad (4)$$

KAPPA-EXP. In [6], the concept of *criticality* was suggested for Go—as some intersections on the board are more important for winning the game than others, these should be recognized as "critical" or "hot", and receive special attention or search effort. To identify critical points, statistics are collected during playouts on which player owns which intersections at the end of each simulation, and on how strongly this ownership is correlated with winning the simulated game. Different formulas have since been suggested to compute the strength of this relationship [6,20]. In the KAPPA-EXP strategy, we use a related concept for identifying not only "hot" intersections from the set of all intersections of a board, but also "hot" boards from the set of all positions in a game. The KAPPA-EXP strategy distributes time proportional to the expected maximum point criticality given the current move number, as estimated from a database of games played by the program itself. The idea is that the maximum point criticality, taken over the set of all intersections I on the board, indicates how crucial the current move choice is. We chose Formula 5 to represent the criticality of an intersection i in move m—the *kappa statistic*, a chance-corrected measure of agreement typically used to

quantify inter-rater reliability [4]. Here, it is employed to quantify agreement between the variables "intersection i is owned by the player at the end of a playout during m's move search" and "the player wins a playout during m's move search".

$$\begin{aligned}\kappa^m(i) &= \frac{\text{agreement}^m_{\text{observed}} - \text{agreement}^m_{\text{expected}}}{1 - \text{agreement}^m_{\text{expected}}} \\ &= \frac{\frac{o^m_{\text{winner}}(i)}{n} - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})}{1 - (o^m_{\text{white}}(i) o^m_{\text{black}}(i) + w^m_{\text{white}} w^m_{\text{black}})}\end{aligned} \tag{5}$$

where n is the total number of playouts, $o^m_{\text{winner}}(i)$ is the number of playouts in which point i ends up being owned by the playout winner, $o^m_{\text{white}}(i)$ and $o^m_{\text{black}}(i)$ are the numbers of playouts in which point i ends up being owned by White and Black, respectively, and w^m_{white} and w^m_{black} are the numbers of playouts won by White and Black, respectively. All numbers refer to the search for move m.

For application at move number m during a game, the average maximum point criticality $\kappa_{\text{avg}} = \frac{1}{g} \sum_{j=1}^{g} \max_{i \in I} \kappa^m_{\text{game j}}(i)$ is precomputed from a database of g games, linearly transformed using parameters for slope and intercept $s_{\kappa_{\text{avg}}}$ and $i_{\kappa_{\text{avg}}}$, and finally multiplied by the search time resulting in Formula 6.

$$t_{\text{nextmove}} = (\kappa_{\text{avg}} \cdot s_{\kappa_{\text{avg}}} + i_{\kappa_{\text{avg}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{6}$$

KAPPA-LM. Instead of using the expected criticality for the current move number as defined above, the KAPPA-LM strategy uses the observed criticality as computed during the search for the player's previous move in the game. This value $\kappa_{\text{lastmove}} = \max_{i \in I} \kappa^{m-2}_{\text{current game}}(i)$ is again linearly transformed using parameters $s_{\kappa_{\text{lastmove}}}$ and $i_{\kappa_{\text{lastmove}}}$, and multiplied with the base search time. The formula is as follows:

$$t_{\text{nextmove}} = (\kappa_{\text{lastmove}} \cdot s_{\kappa_{\text{lastmove}}} + i_{\kappa_{\text{lastmove}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{7}$$

For both KAPPA-EXP and KAPPA-LM, lower and upper bounds for the κ factor ensure reasonable time allocations even in extreme positions.

3.2 Dynamic Strategies

The following five strategies make time-allocation decisions for a move search while the respective search process is being carried out.

BEHIND. As suggested by [13] as the "think longer when behind" heuristic, the BEHIND strategy prolongs the search by a factor f_{behind} if the player is falling behind. It triggers if after the regular search time—as computed

by the semi-dynamic strategies described above—the win rate of the best move at the root is lower than a threshold v_{behind}. The rationale is that by using more time resources, the player could still find a way to turn the game around, while saving time for later moves is less important in a losing position.

UNST. The UNSTABLE strategy, called "unstable evaluation" heuristic in [13], prolongs the search by a factor f_{unstable} if after the regular search time the most-visited move at the root is not the highest-valued move as well. This indicates that by searching longer, a new move could become the most-visited and thus change the final move choice. We have modified this heuristic to check its condition for search continuation repeatedly in a loop. The maximum number of loops until the search is terminated is bound by a parameter l_{unstable}. The single-check heuristic is called UNST-1, the multiple-check heuristic UNST-N in the following.

CLOSE. Similar to a strategy developed independently in [3], the CLOSE strategy prolongs the search by a factor $f_{\text{closesecond}}$ if after the regular search time the most-visited move and the second-most-visited move at the root are "too close", defined by having a relative visit difference lower than a threshold $d_{\text{closesecond}}$. Like the UNST strategy, CLOSE aims to identify difficult decisions that can make efficient use of an increase in search time. In our implementation, this strategy can either be triggered only once (CLOSE-1) or repeatedly (CLOSE-N) after the regular search time is over. For CLOSE-N, a parameter $l_{\text{closesecond}}$ defines the maximum number of loops.

KAPPA-CM. Unlike the three dynamic strategies described above, the KAPPA-CM strategy does not wait for the regular search time to end. Instead, it uses the first, e.g., 100 milliseconds of the search process to collect playout data and then uses the maximum point criticality of the current move $\kappa_{\text{currentmove}} = \max_{i \in I} \kappa^{m}_{\text{current game}}(i)$ to modify the remaining search time. The formula is as follows:

$$t_{\text{currentmove}} = (\kappa_{\text{currentmove}} \cdot s_{\kappa_{\text{currentmove}}} + i_{\kappa_{\text{currentmove}}}) \cdot \frac{t_{\text{remaining}}}{m_{\text{expected}}} \tag{8}$$

The remaining search time can be either reduced or increased by this strategy. Upper and lower limits to the total search time apply.

EARLY. The "early exit" (EARLY-A) strategy, mentioned independently—but not evaluated—in [3], is based on the idea of terminating the search process as early as possible in case the best move cannot change anymore. Therefore, the search speed in playouts per second is measured, and in regular intervals (e.g., 50 playouts) it is checked how many playouts are still expected in the remainder of the total planned search time as determined by the various strategies described above. If the number of playouts needed for the second-most-visited move at the root to catch up to the most-visited one exceeds this expected number of remaining playouts, the search can safely be terminated without changing the final outcome.

If the expected time savings by this strategy are not taken into account when computing planned search times, savings will accumulate throughout the game and early moves cannot benefit from them. In order to achieve a more equal distribution of the resulting time savings among all searches in the game, planned search times can be multiplied with a factor $f_{\text{earlyexit}}$ that is based on average time savings (EARLY-B strategy).

Because in general, not all of the remaining playouts in a search will start with the second-most-visited move, we implemented a parameter $p_{\text{earlyexit}}$ representing an estimate of the proportion of remaining playouts that actually sample the second-most-visited move (EARLY-C strategy). When using this parameter, the search is terminated if the number of playouts needed for the second-most-visited move at the root to catch up to the most-visited one exceeds the expected number of remaining playouts multiplied by $p_{\text{earlyexit}}$. In this case, an unchanged final outcome is no longer guaranteed.

4 Experimental Results

All time-management strategies were implemented in Orego [8] version 7.08. Orego is a Go program using a number of MCTS enhancements like a transposition table [12], RAVE [10], a simulation policy similar to that proposed in [11], and LGRF-2 [2]. The program ran on a CentOS Linux server consisting of four AMD Twelve-Core OpteronT 6174 processors (2.2 GHz). Unless specified otherwise, each experimental run involved 5000 games (2500 as Black and 2500 as White) of Orego against the classic (non-MCTS-based) program GNU Go 3.8 [9], played on the 13×13 board, using Chinese rules (area scoring), positional superko, and 7.5 komi. GNU Go ran at its default level of 10, with the capture-all-dead option turned on. Orego used a single thread and no pondering. Orego used a time limit of 30 seconds per game unless specified otherwise, while Gnu Go had no time limit.

The remainder of this section is structured as follows. In 4.1, the strategies in [13] are tested as a baseline. Next, 4.2 presents results of experiments with semi-dynamic strategies. Dynamic strategies are tested in 4.3. Finally, in 4.4 the best-performing strategy is compared to the baseline in self-play, as well as to Orego with fixed time per move.

4.1 ERICA-BASELINE

In order to compare our results to a state-of-the-art baseline, the strategies described in [13] were implemented and evaluated. The thinking time per move was computed according to the "basic formula"

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{C} \tag{9}$$

as well as the "enhanced formula"

$$t_{\text{nextmove}} = \frac{t_{\text{remaining}}}{C + \max(\text{MaxPly} - \text{MoveNumber}, 0)} \tag{10}$$

Table 1. Performance of ERICA's time management according to [13]

Player	Win rate against GNU GO	95% conf. int.
Basic formula	28.6%	27.3%–29.9%
Enhanced formula	31.4%	30.1%–32.7%
ERICA-BASELINE	34.3%	33.0%–35.7%

where $C = 30$ and MaxPly $= 40$ were found to be optimal values for OREGO. The "unstable evaluation" heuristic, using a single loop as proposed in [13], worked best with $f_{\text{unstable}} = 1$. The "think longer when behind" heuristic, however, did not produce significant improvements for any of the tested settings for v_{behind} and f_{behind}. This seems to be due to differences between OREGO and the ERICA program used in [13]. Preliminary tests showed that positive effects for this heuristic could not be achieved on any board size.

ERICA's time management strategies were tested against GNU GO using the basic formula, using the enhanced formula, and using the enhanced formula plus "unstable evaluation" heuristic (called ERICA-BASELINE from now on). Table 1 presents the results—the enhanced formula is significantly stronger than the basic formula (p<0.01), and ERICA-BASELINE is significantly stronger than the enhanced formula (p<0.01).

4.2 Semi-dynamic Strategies

EXP-MOVES, EXP-SIM and EXP-STONES. As our basic time-management approach, EXP-MOVES, EXP-SIM, and EXP-STONES were tested. The first three rows of Table 2 show the results. As EXP-STONES performed best, it was used as the basis for all further experiments.

OPEN. According to preliminary experiments with OPEN, the "opening factor" $f_{\text{opening}} = 2.5$ seemed most promising. It was subsequently tested against GNU GO. Table 2 shows the result: EXP-STONES with OPEN is significantly stronger than plain EXP-STONES (p<0.001).

MID. Initial experiments with MID showed Formula 3 to perform best with $a = 2$, $b = 40$, and $c = 20$. It was then tested against GNU GO. As Table 2 reveals, EXP-STONES with MID is significantly stronger than plain EXP-STONES (p<0.001).

KAPPA-EXP. The best parameter setting for KAPPA-EXP found in preliminary experiments was $s_{\kappa_{\text{avg}}} = 8.33$ and $i_{\kappa_{\text{avg}}} = -0.67$. Lower and upper bounds for the *kappa* factor were set to 0.5 and 10, respectively. Table 2 presents the result of testing this setting. EXP-STONES with KAPPA-EXP is significantly stronger than plain EXP-STONES (p<0.001).

KAPPA-LM. Here, $s_{\kappa_{\text{lastmove}}} = 8.33$ and $i_{\kappa_{\text{lastmove}}} = -0.67$ were chosen for further testing against GNU GO as well. Lower and upper bounds for the *kappa* factor were set to 0.25 and 10. The test result is shown in Table 2. EXP-STONES with KAPPA-LM is significantly stronger than plain EXP-STONES (p<0.001).

Table 2. Performance of the semi-dynamic strategies investigated

Player	Win rate against GNU GO	95% conf. int.
EXP-MOVES	24.0%	22.9%–25.2%
EXP-SIM	13.0%	12.0%–13.9%
EXP-STONES	25.5%	24.3%–26.7%
EXP-STONES with OPEN	32.0%	30.8%–33.4%
EXP-STONES with MID	30.6%	29.3%–31.9%
EXP-STONES with KAPPA-EXP	31.7%	30.5%–33.0%
EXP-STONES with KAPPA-LM	31.1%	29.9%–32.4%
ERICA-BASELINE	34.3%	33.0%–35.7%

4.3 Dynamic Strategies

BEHIND. We tested all possible combinations of $f_{\text{behind}} = \{0.25, 0.5, 0.75, 1.0, 1.5, 2.0\}$ and $v_{\text{behind}} = \{0.35, 0.4, 0.45\}$. However, just like the "enhanced formula" of [13], EXP-STONES was not found to be significantly improved by BEHIND in OREGO. The best parameter settings in preliminary experiments were $f_{\text{behind}} = 0.75$ and $v_{\text{behind}} = 0.45$. Detailed results are given in Table 3.

UNST. The best results in initial experiments with UNST-1 were achieved by $f_{\text{unstable}} = 1.5$. For UNST-N, $f_{\text{unstable}} = 0.75$ and $l_{\text{unstable}} = 2$ turned out to be promising values. These settings were tested against GNU GO; Table 3 shows the results. EXP-STONES with UNST-1 is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with UNST-N, in turn, is significantly stronger than EXP-STONES with UNST-1 ($p<0.05$).

CLOSE. The best-performing parameter settings in initial experiments with CLOSE-1 were $f_{\text{closesecond}} = 1.5$ and $d_{\text{closesecond}} = 0.4$. When we introduced CLOSE-N, $f_{\text{closesecond}} = 0.5$, $d_{\text{closesecond}} = 0.5$ and $l_{\text{closesecond}} = 4$ appeared to be most successful. Table 3 presents the results of testing both variants against GNU GO. EXP-STONES with CLOSE-1 is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with CLOSE-N, in turn, is significantly stronger than EXP-STONES with CLOSE-1 ($p<0.001$). EXP-STONES with CLOSE-N is also significantly stronger than ERICA-BASELINE ($p<0.05$).

KAPPA-CM. The best parameter setting for KAPPA-CM found in preliminary experiments was $s_{\kappa_{\text{currentmove}}} = 8.33$ and $i_{\kappa_{\text{currentmove}}} = -1.33$. Lower and upper bounds for the *kappa* factor were set to 0.6 and 10. Table 3 reveals the result of testing this setting against GNU GO. EXP-STONES with KAPPA-CM is significantly stronger than plain EXP-STONES ($p<0.05$). However, it is surprisingly weaker than both EXP-STONES using KAPPA-EXP and EXP-STONES with KAPPA-LM ($p<0.001$). The time of 100 msec used to collect current criticality information might be too short, such that noise is too high.

EARLY. First, the EARLY-A strategy was tested. Table 3 presents the result—the improvement to plain EXP-STONES was not significant. Then, we

Table 3. Performance of the dynamic strategies investigated.

Player	Win rate against GNU Go	95% conf. int.
EXP-STONES with BEHIND	25.6%	24.4%–26.9%
EXP-STONES with UNST-1	33.6%	32.3%–34.9%
EXP-STONES with UNST-N	35.8%	34.4%–37.1%
EXP-STONES with CLOSE-1	32.6%	31.3%–33.9%
EXP-STONES with CLOSE-N	36.5%	35.2%–37.9%
EXP-STONES with KAPPA-CM	27.3%	26.1%–28.6%
EXP-STONES with EARLY-A	25.3%	24.1%–26.5%
EXP-STONES with EARLY-B	36.7%	35.4%–38.0%
EXP-STONES with EARLY-C	39.1%	38.0%–40.8%
EXP-STONES	25.5%	24.3%–26.7%
ERICA-BASELINE	34.3%	33.0%–35.7%

introduced $f_{earlyexit}$ in EARLY-B and found a value of $f_{earlyexit} = 2$ to be promising in initial testing. This setting was used against GNU GO in another 5000 games. Finally, $p_{earlyexit}$ was introduced in EARLY-C, which resulted in a change in the best settings found: $f_{earlyexit} = 2.5$ and $p_{earlyexit} = 0.4$ were tested. EXP-STONES with EARLY-B is significantly stronger than plain EXP-STONES ($p<0.001$). EXP-STONES with EARLY-C in turn is significantly stronger than EXP-STONES with EARLY-B ($p<0.01$). This best-performing version is also significantly stronger than ERICA-BASELINE ($p<0.001$).

4.4 Strength Comparisons

Comparison with ERICA-BASELINE on 13×13. Our strongest time-management strategy on the 13×13 board, EXP-STONES with EARLY-C, was tested in self-play against OREGO with ERICA-BASELINE. Time settings of 30, 60 and 120 seconds per game were used with 2000 games per data point. Table 4 presents the results: For all time settings, EXP-STONES with EARLY-C was significantly stronger ($p<0.001$).

Comparison with ERICA-BASELINE on 19×19. In this experiment, we pitted EXP-STONES with EARLY-C against ERICA-BASELINE on the 19×19 board. The best parameter settings found were $C = 80$, MaxPly = 110 and $f_{unstable} = 1$ for ERICA-BASELINE, and $f_{earlyexit} = 2.2$ and

Table 4. Performance of EXP-STONES with EARLY-C vs. ERICA-BASELINE, 13×13 board

Time setting	Win rate against ERICA-BASELINE	95% conf. int.
30 sec sudden death	61.4%	59.2%–63.5%
60 sec sudden death	59.9%	57.7%–62.0%
120 sec sudden death	62.5%	60.4%–64.6%

Table 5. Performance of EXP-STONES with EARLY-C vs. ERICA-BASELINE, 19×19 board

Time setting	Win rate against ERICA-BASELINE	95% conf. int.
300 sec sudden death	62.1%	60.0%–64.2%
900 sec sudden death	59.5%	57.4%–61.7%

$p_{\text{earlyexit}} = 0.45$ for EARLY-C. Time settings of 300 and 900 seconds per game were used with 2000 games per data point. The results are shown in Table 5—for both time settings, EXP-STONES with EARLY-C was significantly stronger ($p<0.001$).

Comparison with fixed time per move. To illustrate the effect of successful time management, two additional experiments were conducted with Orego using fixed time per move in 13×13 Go. In the first experiment, the time per move (650 msec) was set so that approximately the same win rate against GNU Go was achieved as with EXP-STONES and EARLY-C at 30 seconds per game. The result of 2500 games demonstrated that the average time needed per game was 49.0 seconds—63% more than needed by our time-management strategy. In the second experiment, the time per move (425 msec) was set so that the average time per game was approximately equal to 30 seconds. In 2500 games under these conditions, Orego could only achieve a 27.6% win rate, 11.5% less than with EXP-STONES and EARLY-C.

5 Conclusion and Future Research

In this paper, we investigated a variety of time-management strategies for Monte-Carlo Tree Search, using the game of Go as a testbed. Empirical results show that of our proposed strategies, EXP-STONES with EARLY-C and EXP-STONES with CLOSE-N each provide a significant improvement over the state of the art as represented by ERICA-BASELINE in 13×13 Go. For sudden-death time controls of 30 seconds per game, EXP-STONES with EARLY-C increases Orego's win rate against GNU Go from 34.3% to 39.1%. In self-play, this strategy wins approximately 60% of games against ERICA-BASELINE, both in 13×13 and 19×19 Go under various time controls.

Several promising directions remain for future research. We mention three of them. First, a natural direction is the combined testing and optimization of all above strategies—in order to determine to which degree their positive effects on playing strength can complement each other, or to which degree they could be redundant or possibly interfere. First naive attempts at combining strategies have not showed significant improvements. To account for possible interactions, a non-linear classifier like a neural network could be trained to decide about continuing or aborting the search in short intervals, using all relevant information used by above strategies as input. The second direction is to develop enhanced strategies to measure the complexity and importance of a position and thus to use effectively time where it is most needed. Counting the number of independent fights on the board could be one possible approach. Third, the most

successful time-management strategies should be tested in other games or sequential decision problems with time limits in general. CLOSE-N, UNST-N as well as EARLY-C, for example, are domain-independent MCTS enhancements.

Acknowledgment. This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

References

1. Althöfer, I., Donninger, C., Lorenz, U., Rottmann, V.: On Timing, Permanent Brain and Human Intervention. In: van den Herik, H.J., Herschberg, I.S., Uiterwijk, J.W.H.M. (eds.) Advances in Computer Chess, vol. 7, pp. 285–297. University of Limburg, Maastricht (1994)
2. Baier, H., Drake, P.: The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 303–309 (2010)
3. Baudiš, P.: MCTS with Information Sharing. Master's thesis, Charles University, Prague, Czech Republic (2011)
4. Cohen, J.: A Coefficient of Agreement for Nominal Scales. Educational and Psychological Measurement 20(1), 37–46 (1960)
5. 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)
6. Coulom, R.: Criticality: a Monte-Carlo Heuristic for Go Programs. University of Electro-Communications, Tokyo, Japan (2009), Invited talk
7. Donninger, C.: A la recherche du temps perdu: 'That was easy'. ICCA Journal 17(1), 31–35 (1994)
8. Drake, P.: et al.: Orego Go Program (2011), http://legacy.lclark.edu/~drake/Orego.html
9. Free Software Foundation: GNU Go 3.8 (2009), http://www.gnu.org/software/gnugo/
10. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the Twenty-Fourth International Conference on Machine Learning (ICML 2007). ACM International Conference Proceeding Series, vol. 227, pp. 273–280. ACM (2007)
11. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Tech. rep., HAL - CCSd - CNRS (2006)
12. Greenblatt, R., Eastlake III, D., Crocker, S.D.: The Greenblatt Chess Program. In: Proceedings of the Fall Joint Computer Conference, pp. 801–810 (1967)
13. Huang, S.C., Coulom, R., Lin, S.S.: Time Management for Monte-Carlo Tree Search Applied to the Game of Go. In: International Conference on Technologies and Applications of Artificial Intelligence, pp. 462–466. IEEE Computer Society, Los Alamitos (2010)
14. Hyatt, R.M.: Using Time Wisely. ICCA Journal 7(1), 4–9 (1984)
15. 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)

16. Kocsis, L., Uiterwijk, J.W.H.M., van den Herik, H.J.: Learning Time Allocation Using Neural Networks. In: Marsland, T.A., Frank, I. (eds.) CG 2001. LNCS, vol. 2063, pp. 170–185. Springer, Heidelberg (2002)
17. Lee, C.S., Wang, M.H., Chaslot, G.M.J.B., 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 1(1), 73–89 (2009)
18. Lee, C.S., Müller, M., Teytaud, O.: Special Issue on Monte Carlo Techniques and Computer Go. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 225–228 (2010)
19. Markovitch, S., Sella, Y.: Learning of Resource Allocation Strategies for Game Playing. Computational Intelligence 12(1), 88–105 (1996)
20. Pellegrino, S., Hubbard, A., Galbraith, J., Drake, P., Chen, Y.P.: Localizing Search in Monte-Carlo Go Using Statistical Covariance. ICGA Journal 32(3), 154–160 (2009)
21. Šolak, R., Vučković, V.: Time Management during a Chess Game. ICGA Journal 32(4), 206–220 (2009)

An MCTS Program to Play EinStein Würfelt Nicht!

Richard J. Lorentz

Department of Computer Science,
California State University,
Northridge CA 91330-8281, USA
lorentz@csun.edu

Abstract. EinStein Würfelt Nicht! is a game that has elements of strategy, tactics, and chance. Reasonable evaluation functions can be found for this game and, indeed, there are some strong mini-max based programs for EinStein Würfelt Nicht! We have constructed an MCTS program to play this game. We describe its basic structure and its strengths and weaknesses with the idea of comparing it to existing mini-max based programs and comparing the MCTS version to a pure MC version.

1 Introduction

EinStein Würfelt Nicht! (EWN) is a fairly new game, invented in 2004 by Ingo Althöfer [1]. It is played on a 5 × 5 board where each player (called Blue and Red) starts with six pieces, numbered 1 through 6, placed in a triangular pattern as shown in Figure 1.[1] The initial numbering of the pieces is done randomly.

Fig. 1. A typical EWN starting position

Blue moves (tiles are indicated by +) first and the game is won by the first player to place one of his[2] pieces in the opponent's corner square, referred to as the goal. Pieces move one square at a time, always towards the goal, either horizontally, vertically, or diagonally. For example, this means that if a piece is not on one of the edges adjacent to his goal, it will have three legal moves. If a piece moves on top of an existing piece (his own or the opponent's) that piece is captured. The piece to move is decided by the roll of a die. In the case of Figure 1 we see that the die roll is 4 and so Blue can move his piece numbered 4 either to the south, east, or diagonally southeast. If instead a 2 had been rolled then Blue's legal moves would be either to capture his own 4 or 6

[1] All EWN figures are taken from the Little Golem web site [10].
[2] For brevity, we use 'he' and 'his' whenever 'he or she' and 'his or her' are meant.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 52–59, 2012.
© Springer-Verlag Berlin Heidelberg 2012

stone or move diagonally southeast. If the die rolled corresponds to a stone that has been captured then we find the lowest numbered stone greater than the die value and the highest numbered stone less than the die value and we may move either of these stones. An alternate way to win the game is to capture all of the opponent's pieces.

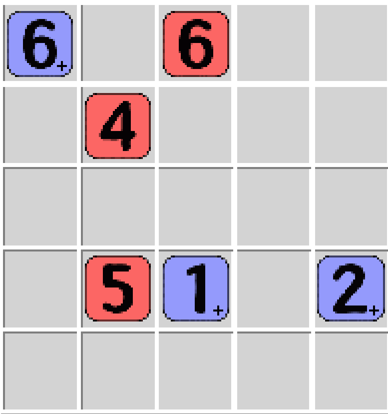


Fig. 2. Late in an EWN game

Consider the position shown in Figure 2. If it is Blue's turn to move, since his 3 piece is not on the board he may move either piece 2 or piece 6. There are three legal moves for piece 6, including the capture of Red's piece 4, but more interesting there is only one legal move for Blue's piece 2 and that move lands in the goal so Blue can win the game. If it is Red's turn to play he is only allowed to move piece 4, but among his three legal moves one of them is to capture Blue's 6 on the goal square, winning for Red.

EWN has both tactical and strategic elements. Near the end of the game a player must carefully calculate which pieces are likely to reach the goal square while being careful to not allow all of his pieces to become captured. But these tactical decisions require probabilistic calculations because the die determines which pieces move. Throughout the game there are a number of competing strategic ideas. Clearly having a piece near the goal increases the chances of reaching the goal. However, if the pieces numerically adjacent to the piece near the goal are still on the board then there is only a one in six chance that piece will be able to move so it is desirable to have some of your own stones captured (either by your opponent or by yourself) to increase the chance of being able to move pieces that are near the goal. This needs to be balanced with the need to prevent all of ones pieces from being captured.

This mix of strategy, tactics, and randomness, combined with the fact that games are not too long makes EWN a fun and interesting game to play.

2 ONESTONE, an EWN Playing Program Test Bed

We have written an MCTS-based program, named ONESTONE, to play EWN. It was created as an experiment to see how well the MCTS paradigm works with a simple game with random elements. Intuitively, this should be an ideal setting since Monte-Carlo simulations should be an effective way to deal with the randomness in the game. In fact, one of the questions we hoped to answer was whether an MCTS version will outperform a basic Monte-Carlo player. It turns out that it does, but not by very much. We were also interested in seeing how an MCTS program could compete against existing mini-max based programs. It appears to play on a par with such programs.

There are a number of existing EWN playing programs that we are aware of. Some are inactive now and most have and/or do play via the Little Golem website (LG) [8]. Fig. 3 shows a summary of these programs.

Testing ONESTONE was done via self-tests and LG. EWN has a lively presence on LG with its many human players and five other programs that play fairly regularly. However, this is a "turn based" game-playing site which means that players have considerable time to make their moves, usually 24 hours or more. This is not a particularly expeditious vehicle for testing a developing program. So, we decided to develop first a very basic version and then use that version as the basis for self-testing future versions. Finally, we took our most sophisticated version of ONESTONE and let it play on LG to see how it faired against the other programs and human players.

2.1 The Road to UCT

As we said above, we constructed the program incrementally. First we built a basic Monte-Carlo program that would play the game and then made a straightforward play UCB type enhancement [2] so that the more promising moves would be simulated more often. This version became our standard from which we tested all future versions. We found that even with this straightforward UCB construction ONESTONE played a reasonably strong game.

Program name	Author	Status	Program type
FRAGGLE [9]	Ingo Schwab	inactive	mini-max with endgame tablebases
GAMBLER	Richard Pijl	active on LG	mini-max, in development
HANFRIED	Stefan Schwarz and Jörg Sameith	active on LG	mini-max
MEINSTEIN	Theo van der Storm	inactive[3]	mini-max
NAÏVE CHILD	Mark Pawlenka	active on LG	pure MC (no MCTS tree)
RORORO THE BOT	Phil Carmody	active on LG	mini-max, emphasis on speed
SYBIL	Wesley Turner	active on LG	mini-max with endgame tablebases

Fig. 3. EWN playing programs

The next step was to add the tree structure for MCTS. We did this using the basic UCT algorithm [3,5,7]. But we needed to add additional structure to deal with the die throws. A typical node in a normal UCT tree will have elements corresponding to the visit counts, the win counts, and a pointer to the children. A node in the UCT tree for ONESTONE still contains a single visit count and a single win count, but also contains six separate child pointers. Conceptually, from any node we are creating six different UCT subtrees, one for each possible die throw. Since each child has an equal chance of being the selected move, during UCT expansion we make sure the children are

[3] MEINSTEIN competed in this year's Computer Olympiad in memory of the late program's author. It was operated by Jan Krabbenbos,

visited uniformly and then collect the visits and wins for all six children of a node into a single value.

There is a temptation to emphasize die throws corresponding to subtrees that are less well understood, e.g., that have wins/visit ratios near 50%. Similarly, as pieces start leaving the board multiple die throws will correspond to the same choice of moves and it is tempting to try to collect these into a single search. In both cases, however, there is no easy way to accomplish this while retaining the proper wins/visit ratio for the subtree's parent.

There are two natural ways to visit the children uniformly. Any time we visit a node in the tree we can randomly choose the child. In some ways this seems very much in the spirit of Monte-Carlo simulation and random die throwing. Nevertheless, we elected to do it more methodically, by keeping track in every node the last child visited and then visiting the next child when that node is visited again.

2.2 MCTS Improvements

There are a number of techniques used for improving MCTS performance, many of which were developed for and have been extremely successful with Go-playing programs. We implemented two of the most common ones in ONESTONE.

The first is to improve the quality of the random playouts, a suggestion that was originally articulated in [6]. The point of a random playout is to give some measure of the strength of a position, or more precisely the likelihood the player will actually win from that position. This means that the closer the playout looks like a real game, the more information it should be providing. But it is well known that improving the playouts must be done with great care. In fact, somewhat counter intuitively, reasonable looking playout "improvements" can actually make the program play worse. This is because the improved moves can introduce subtle bias that, for example, might encourage certain move sequences that are not always favorable.

We made two major modifications to our playouts. The first modification is to look for an immediately winning move (a "mate-in-one") and playing it if it is available. It does not make much sense to continue a random playout if a player is sitting on a winning move but is not playing that move. The second modification to the playouts encouraged moves that either get closer to the goal or that capture a piece. For example, in the position in Figure 2 if a 5 is the die value, it seems unlikely that moving piece 5 west is the best move. Moving north or northwest is much more likely to be the best move. For this reason, during a random playout we make it twice as likely that either of the two suggested moves is made over the move to the west. Similarly, captures, whether of an enemy piece or your own, are often interesting. Like moves towards the goal, captures are also given double the chance of being made during a random playout.

The second MCTS improvement that we made is to give "prior" initial values to new MCTS tree nodes. When creating new nodes for the MCTS tree the win counts and the total playout counts are usually initialized to zero. However, if there is prior knowledge of the quality of the move represented by a node relative to its siblings, then it sometimes makes sense to initialize these values to something other than zero

that reflects the perceived relative value of that position [4]. In OneStone we used the same ideas here that we used for the random playouts. New MCTS tree nodes are initialized to have visit counts of 100. The win counts of moves that move towards the goal are scaled. A move that places a piece one square away from the goal is given an initial win count of 90. If two away from the goal, 75. Three away, 60. Four away gets the default value of all moves, 40, and 5 away from the goal gets a value of 20. We also continue to assume a capture is interesting, so a capture is given an initial value of 65. If a move is both a capture and a move towards the goal the initial win count is set to the maximum of the two plus 5.

3 EWN Variants

There are a number of variants to EWN, two of which are also played on LG. One is referred to as “Black Hole”. In this variant the square in the middle of the board is designated as the black hole in that any piece that moves to that square is immediately removed from the board. Since it is often strategically sound to capture one’s own pieces to increase the strength of the remaining pieces, this provides another means to reduce the number of one’s own pieces on the board.

The other variant is called “Backwards Capture” and it has the property that when capturing one may capture any piece adjacent to one’s own, orthogonally or diagonally, that is, in any of eight possible different directions. As will be discussed in a bit more detail below, the black hole variant appears not to change the complexity of the game very much, while backwards capture seems to make the game quite a bit different. For these experiments, our implementations for the two variants are essentially equivalent to the EWN implementation. That is, that random playout biases and the prior value settings are set the same in the variants as they are in the normal EWN game. Certainly this is not optimal for the two variants but it does give us a good feel for how the variants differ in this early stage of program development.

4 Results

We compared the MCTS version of the program against the basic MC version of all three variants. Each variant played 500 games as Blue, that is, moving first, and 500 games playing Red. Each player is given 30 seconds per move. The results are summarized in Table 4 where the MCTS results are shown in bold.

In our experiments we found that when playing one version against itself the blue player had a slight edge. This effect can be seen in the table where we notice that

	MCTS playing Blue	MCTS playing Red
Normal EWN	**291** – 209	230 – **268**
Black Hole	**288** – 212	238 – **262**
Backwards capture	**416** – 84	101 – **399**

Fig. 4. Results of MCTS vs. plain MC for all three variants

though the MCTS version always outperforms the MC version, when playing Blue the improvement is even greater. But the critical observation is that the MCTS versions always do outperform the basic MC version, though maybe not to the levels we had hoped.

The third line of the table we find to be quite remarkable. From the fact that the MCTS version is so much stronger than the basic MC version we may conclude that the backwards capture variant is, in some sense, a more complicated game than the other two variants and thus the MC approach is insufficient to deal with the subtleties of this variant. Here is an example of a fairly simple game where MCTS is shown to be substantially superior to Monte-Carlo.

We let our best versions of ONESTONE play on LG. On LG most of the contests are multiple game matches where the first to win 3 (or 5 or 7) games wins the match. This, in some sense, removes the first move advantage because the winner of any game in a match will move second in the next game. Since games progress so slowly on LG the data is still a bit thin, but the Tables 5 and 6 summarize the current situation. With LG games we allow ONESTONE a maximum of 5 minutes per move but in practice most moves are completed in under a minute. This will be discussed in Section 5. The opponents, of course, have more than 24 hours to make their moves but I suspect most spend about the same amount of time on their moves as ONESTONE does.

EWN variant	wins	losses
Normal	346	120
Black Hole	54	16
Backwards Capture	65	25

Fig. 5. OneStone results on Little Golem

Program	wins	losses
GAMBLER	14	14
HANFRIED	0	0
NAÏVE CHILD	14	4
RORORO THE BOT	5	2
SYBIL	1	0

Fig. 6. OneStone versus other programs

Fig. 5 shows that ONESTONE is playing a respectable game against the mostly human opponents. It has played a few games against other computer programs as shown in Fig. 6. Though there is still not a large amount of data, it appears to be playing at least even with the mini-max based programs and, not surprisingly, is doing better against the pure MC program NAÏVE CHILD.

LG provides ratings for players of its various games. Currently ONESTONE is rated 1777, placing it 15th among the approximately 400 players. Though its rating seems to average around this general value, it is interesting to note that it shows a huge degree of variation. In the past month its rating has been as low as 1625 placing it in position 100 on the overall list and it has been as high as 1880 where it was rated 3rd on the

list. Such huge swings are not uncommon with EWN ratings indicating that the random element of the game is not insignificant.

5 Remarks

We have demonstrated that the MCTS approach to programming EWN is reasonable. Its relative success on LG shows that it is playing a quite good game. Also, the improvements provided by adding a UCT tree show that MCTS provides benefits over pure MC.

The obvious next step would be to tune the biases in the random playouts and the initial prior values in the MCTS tree. Though we do not doubt that there will be some gain in such an endeavor, early experiments seem to indicate that the improvement is likely to be minor. We find ourselves searching for other techniques. Perhaps our notion of "good moves" needs to be refined. Local properties of the piece being moved may need to be taken into account, similar to the way patterns are used in many MCTS Go-playing programs.

A second issue that needs attention is relating the wins/visits ratio with the expected win value of a node. In normal MCTS we know that the wins/visits ratio does not correlate very well with the likelihood of actually winning the game. This is not an issue with normal MCTS programs since, whatever the wins/visits ratio means, that is the value we must use when traversing the UCT tree and deciding which children to visit. In the case of EWN, things are not so clear. Certainly the best move to make corresponds to the node with the highest expected value. But will this always coincide with the node with the highest wins/visits ratio? For example, if some children of a node are reporting wins/visits ratios of 90% the reality is that those moves are almost surely wins. By "almost surely" we mean with much greater probability than 90%. More than likely, the expected value of such a node is actually greater than what is reported by the wins/visits ratio. We would like to understand this better and, ideally, adjust our move selection process accordingly.

The third issue is somewhat related to the comment above. Our experience shows that at the root of the MCTS tree if one node is visited sufficiently more often than are its siblings, there is very little chance another node will catch up any time soon. As a result we find that if we set a cut-off value for this difference and stop the search if this cut-off is reached, the search can be terminated early with no ill effects. Since ONESTONE is playing against humans on LG (and certainly can be made to play locally, too), early termination provides the benefit that moves can often be made quickly, a feature quite desirable to most humans when playing. Tests show that this does no harm to the playing strength of the program, yet with the proper cut-off value we now have ONESTONE making most of its moves in under 30 seconds, about 90% in under 1 minute, and a very few, fewer than 1%, requiring the full 5 minutes allotted. This is a useful practical feature.

The fourth issue is a sequel on the third issue. Because of the discovery mentioned in the previous paragraph we thought it might be beneficial to extend this technique to all nodes in the tree. The idea being that if the visits value of one child of a node is

sufficiently ahead of its siblings we discontinue any searches of the children. Of course, this is an unsafe operation, but it may have value in practice. We experimented with various approaches, such as uniformly using the same cut-off throughout, graduating the cutoff according to the node's depth in the tree, etc. Most approaches did neither harm, nor did they seem to benefit the program.

References

1. Althöfer, I.: On the origins of EinStein würfelt nicht!, `http://www.althofer.de/origins-of-ewn.html`
2. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite time analysis of the multiarmed bandit problem. Machine Learning 47(2-3), 235–256 (2002)
3. 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)
4. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML 2007: Proceedings of the 24th International Conference on Machine Learning, pp. 273–280. ACM, New York (2007)
5. Gelly, S., Wang, Y.: Exploration exploitation in go: UCT for Monte-Carlo Go. In: Twentieth Annual Conference on Neural Information Processing Systems (2006)
6. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France (2006)
7. 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)
8. `http://www.littlegolem.net/jsp/index.jsp`
9. `http://datendissertationschwab.de/4.html`

Monte-Carlo Tree Search Enhancements for Havannah

Jan A. Stankiewicz, Mark H.M. Winands, and Jos W.H.M. Uiterwijk

Department of Knowledge Engineering, Maastricht University
j.stankiewicz@student.maastrichtuniversity.nl,
{m.winands,uiterwijk}@maastrichtuniversity.nl

Abstract. This article shows how the performance of a Monte-Carlo Tree Search (MCTS) player for Havannah can be improved by guiding the search in the playout and selection steps of MCTS. To improve the *playout* step of the MCTS algorithm, we used two techniques to direct the simulations, Last-Good-Reply (LGR) and N-grams. Experiments reveal that LGR gives a significant improvement, although it depends on which LGR variant is used. Using N-grams to guide the playouts also achieves a significant increase in the winning percentage. Combining N-grams with LGR leads to a small additional improvement. To enhance the *selection* step of the MCTS algorithm, we initialize the visit and win counts of the new nodes based on pattern knowledge. By biasing the selection towards joint/neighbor moves, local connections, and edge/corner connections, a significant improvement in the performance is obtained. Experiments show that the best overall performance is obtained when combining the visit-and-win-count initialization with LGR and N-grams. In the best case, a winning percentage of 77.5% can be achieved against the default MCTS program.

1 Introduction

Recently a new paradigm for game-tree search has emerged, the so-called Monte-Carlo Tree Search (MCTS) [6,13]. It is a best-first search algorithm that is guided by Monte-Carlo simulations. In the past few years MCTS has substantially advanced the state-of-the-art in several deterministic game domains where $\alpha\beta$-based search [12] has had difficulties, in particular computer Go [15], but other domains include General Game Playing [3], LOA [25] and Hex [1]. These are all examples of game domains where either a large branching factor or a complex static evaluation function do restrain $\alpha\beta$ search in one way or another.

A game that has recently caught the attention of AI researchers is Havannah, regarded as one of the hardest connection games for computers [24]. Designing an effective evaluation function is quite hard and the branching factor is rather large, making MCTS the algorithm of choice. A substantial amount of research has been performed for applying MCTS in Havannah [16,19,23,24], but humans are still superior. In this article[1] we therefore investigate how the performance

[1] This article is based on the research performed by the first author for his M.Sc. thesis [21].

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 60–71, 2012.
© Springer-Verlag Berlin Heidelberg 2012

of our MCTS-based Havannah program [8,11,21] can be improved by enhancing the playout and selection steps. For the playout step we propose to apply the Last-Good-Reply policy [2,7] and N-grams [14,20]. For the selection step, we bias the moves by using prior knowledge [10] based on patterns.

The article is organized as follows. In Section 2 we explain the rules of Havannah. Next, Section 3 discusses the application of MCTS to Havannah and describes our enhancements for the playout and selection steps. Subsequently, the enhancements are empirically evaluated in Section 4. Finally, in Section 5 we conclude and give an outlook on future research.

2 The Rules of Havannah

Havannah is a turn-based two-player deterministic perfect-information connection game invented by Christian Freeling in 1976 [9]. It is played on a hexagonal board, often with a *base* of 10, meaning that each side has a length of 10 cells. One player uses white stones; the other player uses black stones. The player who plays with white stones starts the game. Each turn, a player places one stone of his color on an empty cell. The goal is to form one of the following three possible winning connections (also shown in Fig. 1).

- **Bridge:** A connection that connects any two corner cells of the board.
- **Fork:** A connection that connects three sides. Corner cells do not count as side cells.
- **Ring:** A connection that surrounds at least one cell. The cell(s) surrounded by a ring may be empty or occupied by white or black stones.

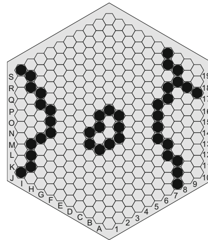


Fig. 1. The three possible connections to win the game. From left to right: a bridge, a ring and a fork.

Because White has an advantage being the starting player, the game is often started using the swap rule. One of the players places a white stone on the board after which the other player may decide whether he[2] will play as White or Black. It is possible for the game to end in a draw, although this is quite unlikely.

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

3 Havannah and Monte-Carlo Tree Search

MCTS is a best-first search algorithm that combines Monte-Carlo evaluation (MCE) with tree search [6,13]. We assume the MCTS algorithm to be known by the readers. For more details we refer to the literature, notably the Ph.D. thesis by Chaslot [4]. In this section we first describe previous MCTS research related to Havannah (3.1), then give our MCTS enhancements for the play-out (3.2) and selection (3.3) steps.

3.1 MCTS Refinements for Havannah

Teytaud and Teytaud [24] introduced MCTS based on UCT in Havannah. Their experiments showed that the number of play-outs per move has a significant impact on the performance. Additions such as Progressive Widening [5] and Rapid Action Value Estimates (RAVE) [10] were used as well. The former gave a small improvement in the winning rate, while RAVE significantly increased the percentage of games won. An enhancement of RAVE (called PoolRave) applied to Havannah gave a further small increase in the winning rate [19]. The idea of adding automatically generated knowledge in the play-out step to guide simulations was first explored by Rimmel and Teytaud [18]. This was dubbed *contextual Monte-Carlo* (CMC) simulation and was based on a reward function learned on a tiling of the simulation space. Experiments for Havannah showed a winning rate of 57% against a program without CMC.

More important was the application of *decisive moves* during the selection and play-out step: whenever there is a winning move, that move is played regardless of the other possible moves. Experiments showed winning percentages in the range of 80% to almost 100% [23]. Fossel [8] used Progressive History [17] and proposed Extended RAVE to improve the selection strategies, giving a winning percentage of approximately 60%.

Several more enhancements for an MCTS player in Havannah were discussed by Lorentz [16]. One is to try to find moves near stones already on the board, thus avoiding playing in empty areas. Another is to recognize forced wins and losses, called Havannah-Mate, which can save time during the search process. A third enhancement is the use of the Killer RAVE heuristic, where only the most important moves are used for computing RAVE values. Each of these enhancements caused a significant increase in the winning percentage.

3.2 Enhancing the Play-out Step in MCTS

This subsection discusses the Last-Good-Reply policy and N-grams which may improve the play-out step of MCTS in Havannah.

Last-Good-Reply. The Last-Good-Reply (LGR) policy [2,7] is an enhancement used during the play-out step of the MCTS algorithm. Rather than applying the default simulation strategy, moves are chosen according to the last good replies to previous moves, based on the results from previous play-outs.

It is often the case in Havannah that certain situations are played out locally. This means that if a certain move is a good reply to another move on some board configuration, it will likely be a good reply to that move on different board configurations as well, because it is only the local situation that matters. However, MCTS itself does not 'see' such similar local situations. The goal of LGR is to improve the way in which MCTS handles such local moves and replies.

There are several variants of LGR [2]. The first one is LGR-1, where each of the winner's moves made during the play-out step is stored as the last good reply to the previous move. During the play-out step of future iterations of the MCTS algorithm, the last good reply to the previous move is always played instead of a (quasi-)random move, whenever possible. Otherwise, the default simulation strategy is used. As an example, consider Fig. 2, where a sequence of moves made during a play-out is shown.

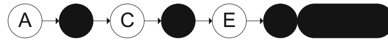


Fig. 2. A simulation in MCTS

Because Black is the winner, the LGR-1 table for Black is updated by storing every move by Black as the last good reply to the previous one. For instance, move B is stored as the last good reply to move A. If White will play move A in future play-outs, Black will always reply by playing B if possible.

The second variant of LGR is LGR-2. As the name suggests, LGR-2 stores the last good reply to the previous two moves. The advantage of LGR-2 is that the last good replies are based on more relevant samples [2]. During the play-out, the last good reply to the previous *two* moves is always played whenever possible. If there is no last good reply known for the previous two moves, LGR-1 is tried instead. Therefore, LGR-2 also stores tables for LGR-1 replies.

A third variant is LGR-1 with forgetting, or simply LGRF-1. This works exactly the same as LGR-1, but now the loser's last good replies are deleted if they were played during the play-out. Consider Fig. 2 again, where White lost the game. For instance, if move C was stored as the last good reply to B for White, it is deleted. Thus, the next time Black plays B, White will chose a move according to the default simulation strategy.

The fourth and last variant is LGRF-2, which is LGR-2 with forgetting. Thus, the last good reply to the previous two moves is stored and after each play-out, the last good replies of the losing player are deleted if they have been played.

N-grams. The concept of N-grams was originally developed by Shannon [20], where he discussed how one can predict the next word, given the previous $N-1$ words. Typical applications of N-grams are, e.g., speech recognition and spelling checkers, where the previous words spoken or written down can help to determine what the next word should be. However, N-grams are also applicable in the context of games, as shown by Laramée [14]. They can be used as an enhancement to the play-out step of the MCTS algorithm. The idea is somewhat similar to LGR. Again, moves are chosen according to their predecessor, but instead of

choosing the last successful reply, the move with the highest winning percentage so far among all legal moves is chosen. Thus, for each legal move i, the ratio $w_{i,j}/p_{i,j}$ is calculated, where $w_{i,j}$ is the number of times playing move i in reply to move j led to a win and $p_{i,j}$ is the number of times move i was played in reply to move j. In order not to make the search too deterministic, the moves are chosen in an ϵ-greedy manner [22]. With a probability of $1 - \epsilon$ an N-gram move is chosen, while in all other cases, a move is chosen based on the default simulation strategy. Furthermore, the values $w_{i,j}$ and $p_{i,j}$ in the N-gram tables are multiplied by a decay factor γ after every move played in the game, where $0 \leq \gamma \leq 1$. This ensures that, as the game progresses, new moves will be tried as well, instead of only playing the same N-gram moves over and over again.

It is also possible to combine N-grams with a certain threshold T. The reason to apply thresholding, is to try to improve the reliability of N-grams. The more often a certain N-gram has been played, the more reliable it is. If the N-gram of a proposed move has been played fewer than T times before, the move is not taken into consideration. If all of the available moves have been played fewer than T times, the default simulation strategy is applied.

Like with LGR, N-grams can be extended to take into account the previous two moves, instead of only the previous move. To distinguish between the two, ‘N-gram1’ refers to N-grams based on only the previous move, while ‘N-gram2’ refers to N-grams based on the previous two moves.

Because N-gram1 and N-gram2 are based on different contexts, combining the two may give a better performance than using N-gram1 or N-gram2 separately. N-gram1 and N-gram2 can be combined using averaging. Rather than choosing moves based purely on N-gram2, the moves are chosen based on the average of the ratios $w_{i,j}/p_{i,j}$ of N-gram1 and N-gram2.

3.3 Enhancing the Selection Step in MCTS

Gelly and Silver [10] proposed the use of prior knowledge for the selection step of the MCTS algorithm, where the visit and win counts of the new nodes are initialized to certain values, based on the properties of the move to which the node corresponds. It is basically an adaptation of the UCT formula, as shown in Equation 1.

$$k \in argmax_{i \in I} \left(\frac{v_i n_i + \alpha_i}{n_i + \beta_i} + C \cdot \sqrt{\frac{\ln n_p}{n_i + \beta_i}} \right) \tag{1}$$

The additional parameters α_i and β_i are the win count bias and visit count bias, respectively, which are based on the properties of the move corresponding to node i. By adding such biases to the win and visit counts of MCTS nodes, the selection can be biased towards certain moves.

This subsection discusses three heuristics how the values of α_i and β_i can be chosen. First, we describe how the selection can be biased towards joint moves and neighbor moves. Then we discuss how local connections can be used to guide the selection. Finally, we describe how the selection can be biased towards edge and corner connections. See Fig. 3 for an overview.

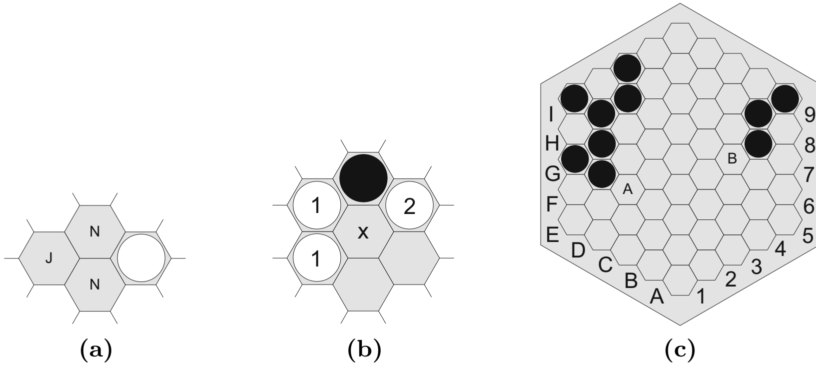


Fig. 3. Biasing the selection towards certain types of moves: joint and neighbor moves (a), local connections (b), and edge and corner connections (c)

Joint and Neighbor Moves. Lorentz [16] proposed to initialize the visit and win counts of new nodes in such a way that the selection is biased towards 'joint moves' and 'neighbor moves'. Joint moves are moves that are located two cells from a stone of the current player and where the two cells between are empty. An example is shown in Fig. 3a, where the joint move is marked by 'J', viewed from White's perspective. Neighbor moves are simply moves adjacent to a cell of the same color, marked by 'N' in Fig. 3a.

Local Connections. A second idea to initialize visit and win counts of new nodes is to take into account the number of connections with existing groups. After all, Havannah is a game in which connections play an important role. As an example, consider Fig. 3b. Assuming that White considers playing at cell '×', there are two local groups of white surrounding stones. They are indicated by the number of the group to which the stone belongs. The stones marked by '1' belong to the same group, as they are connected with each other. The stone marked by '2' is a separate group. If one would play move '×', it would thus form a connection between two local groups. Of course, it could be the case that the two groups are actually connected outside this local area, in which case playing move '×' would simply complete a ring.

Edge and Corner Connections. A third option is to count the number of edges and corners a proposed move would be connected to. The idea is to try to direct the search towards the formation of forks and bridges. For example, see Fig. 3c. Move 'A' on cell E3 connects to one corner and two edges, whereas move 'B' on cell D6 only connects to one corner. Move 'A' is therefore likely to be better than move 'B'. Our MCTS engine already keeps track to which chain each stone belongs [11]. It means that the only additional calculations are (1) to determine which chains the proposed move is connected to and (2) to check for edge and corner cells whether they belong to one of those chains as well.

4 Experiments

This section presents the experiments to assess the enhancements described in the previous section. After describing the experimental setup (4.1), we report the results of experiments with Last-Good-Reply (4.2) and N-grams (4.3). Finally, experiments with several visit-and-win-count initializations (4.4) are discussed.

4.1 Experimental Setup

The experiments discussed in this section were performed on a 2.4 GHz AMD Opteron CPU with 8 GB RAM. Both players use UCT/RAVE with UCT constant C (see Equation (1)) set to 0.4 and RAVE constant R (see [23]) set to 50, plus Havannah-Mate. The default simulation strategy is a uniform random playout. Every experiment was done in a self-play setup. Unless stated otherwise, each experiment consists of 1,000 games on a base-5 board, with a thinking time of 1 second. Because White has a starting advantage, each experiment is split into two parts. During the first 500 games, the enhancements investigated are applied to White and during the second 500 games, they are applied to Black. Throughout the experiments confidence intervals of 95% are used.

4.2 Last-Good-Reply

The performance of Last-Good-Reply was tested with the default setup described in Subsection 4.1. For this first set of experiments, the contents of the LGR tables of the previous turn were used as the initial LGR tables for the current move. The results for all four LGR variants are shown in Table 1.

Table 1. Performance of the four LGR variants

	White	Black	Average
LGR-1	65.8%	49.4%	57.6% (±3.1)
LGR-2	62.4%	47.8%	55.1% (±3.1)
LGRF-1	65.8%	58.0%	61.9% (±3.0)
LGRF-2	59.0%	49.6%	54.3% (±3.1)

As the table shows, LGR generally improves the performance of the MCTS engine. LGR-1 and LGR-2 seem to perform equally well given the confidence intervals, with winning percentages of 57.6% and 55.1%, respectively. Forgetting poor replies seems to give a slight improvement when added to LGR-1, but when added to LGR-2, the performance appears to be the same. LGRF-1 gives a winning percentage of 61.9% while LGRF-2 only wins 54.3% of the games.

Quite remarkably, LGR-2 and LGRF-2 do not perform better than LGR-1 and LGRF-1. In particular, the difference between the performance of LGRF-2 and LGRF-1 is quite significant. It appears that taking a larger context into account when using last good replies, does not lead to a better performance of the MCTS engine. As an additional experiment we tested whether emptying

the LGR table after every move would influence the performance. It turned out, however, that resetting the tables does not have any influence on the performance of the MCTS player. Each of the results lies within the confidence interval of its no-reset equivalent.

4.3 N-grams

N-grams were tested with the default setup described in Subsection 4.1. We first summarize three experiments for configuring the N-grams. For detailed results, see [21]. Next, we investigate the combination of N-grams with LGR policies.

N-gram Configuration Experiments. The first experiment was to determine the best value of the decay factor γ. The N-grams were chosen using ϵ-greedy selection, with $\epsilon = 0.1$. As it turned out, the best value for γ appears to be 0, which means that the N-gram tables are completely reset for each turn. The resulting winning percentages for N-gram1 and N-gram2 are $60.2 \pm 3.0\%$ and $61.3 \pm 3.0\%$, respectively.

In the second set of experiments with N-grams the influence of thresholding was evaluated. Thus, a move was only considered if its N-gram had been played more than T times before. In the case of N-gram2, if the N-gram was played fewer than T times, the N-gram1 of the move was considered. A decay factor $\gamma = 0$ was used. It turned out that thresholding has no positive influence on the performance. In fact, the higher the threshold, the lower the performance seems to get for each of the N-gram variants.

The third set of experiments was performed to determine the influence of averaging N-gram1 and N-gram2, rather than using only N-gram2. Again, the experiment was run with $\gamma = 0$ for different threshold values. The result was that using the average of both N-grams instead of only N-gram2 generally does not give a significant improvement. Again, applying a threshold only decreases the performance. When no threshold is applied, the result is within the confidence interval of the 61.3% result (i.e., the first experiment).

N-grams Combined With LGR. The fourth set of experiments evaluated the performance when N-grams are combined with Last-Good-Reply. The moves in the play-out are chosen as follows. First, the last good reply is tried. If none exists or if it is illegal, the move is chosen using N-grams with a probability of 0.9, thus $\epsilon = 0.1$. Otherwise, the default simulation strategy is applied. Again, the decay factor was set to $\gamma = 0$. No thresholding or averaging was applied to the N-grams. The results are shown in Table 2.

As the table shows, combining LGR with N-grams gives overall a better result. Given the confidence intervals, there seems to be little difference between the performances of the different combinations of LGR and N-grams. For the remainder of the experiments we choose the combination LGRF-2 with N-gram1.

Table 2. The winning percentages of N-grams with and without LGR variants

	N-gram1	N-gram2
no LGR	60.2% (±3.0)	61.3% (±3.0)
LGR-1	62.0% (±3.0)	65.0% (±3.0)
LGR-2	61.0% (±3.0)	60.2% (±3.0)
LGRF-1	62.9% (±3.0)	65.6% (±2.9)
LGRF-2	65.9% (±2.9)	62.2% (±3.0)

4.4 Initializing Visit and Win Count

To test the performance when visit-and-win-count initialization is used in the selection step of MCTS, four sets of experiments were constructed. All results are summarized in Table 3 at the end of this subsection.

Joint and Neighbor Moves. The influence of biasing the selection towards joint and neighbor moves was tested using the default setup, with the required parameters set as follows. All new nodes were initialized with a visit count of 40. Joint moves were given a win count of 30, neighbor moves a win count of 40, and all other moves a win count of 5.

Biasing the selection towards joint and neighbor moves improves the performance considerably, with a winning percentage of 69.2%, which is close to the 67.5% that Lorentz [16] achieved. The experiment was repeated with the addition of LGRF-2 and N-gram1 to the play-out step, increasing the performance significantly to 77.5%. The idea of biasing towards joint moves and neighbor moves was also extended to the play-out step. However, it turned out that this decreases the performance. The reason for this decrease in performance is most likely the computational cost of determining whether a cell corresponds to a joint or neighbor move.

Local Connections. The experiments with biasing the selection towards local connections, were run using the default setup, with the parameter values set as follows. All nodes were initialized with a visit count of 30. Nodes of which the corresponding move was connected to 0, 1, 2, or 3 surrounding groups were given initial win counts of 0, 10, 20, and 30, respectively.

For this initialization scheme, a winning percentage of 61.3% is achieved. This performance is somewhat less than when biasing towards joint and neighbor moves. Again, the experiments were repeated with the addition of LGRF-2 and N-gram1, increasing the winning percentage to 70.0%. However, biasing the selection towards joint and neighbor moves still performs significantly better.

Edge and Corner Connections. A third set of experiments was performed with biasing the selection towards edge and corner connections. For these experiments, the initial visit and win counts were set as follows. If a proposed move would be connected to more than 1 corner or more than 2 edges, the initial visit and win counts were set to 1 and 1000, respectively. In all other cases, the initial

visit count was set to 30, while the initial win count was set to 10 times the number of edges or corners to which the proposed move would be connected.

For this scheme, a winning percentage of 58.7% is achieved, slightly smaller than biasing towards joint and neighbor moves or local connections. When LGRF-2 and N-gram1 is added, the performance increased significantly to 68.3%.

Combination. The fourth set of experiments with visit-and-win-count initialization was based on a combination of the three heuristics described above. This was done in a cumulative way. The initial visit count for each node was set to 100, which is the combined initial visit count of the three heuristics. The initial win count was determined by combining the relevant initial visit counts of the three heuristics. Nodes of which the move would be connected to more than 1 corner or 2 edges, were again given an initial visit count of 1 and a win count of 1000. Combining the three heuristics gives good results (73.4% winning percentage). The combination works better than any of the three heuristics on their own. Adding LGRF-2 and N-gram1 again increases the performance, although the increase is not as large as with the three heuristics individually. In fact, the addition of LGRF-2 and N-gram1 performs just as well as the first heuristic, where the selection is biased towards joint and neighbor moves (77.5%).

Table 3. Biasing the selection towards certain types of moves, without and with the addition of LGRF-2 and N-gram1

	without LGRF-2 + N-gram1	with LGRF-2 + N-gram1
joint and neighbour moves	69.2% (±2.9)	77.5% (±2.6)
local connections	61.3% (±3.0)	70.0% (±2.8)
edge and corner connections	58.7% (±3.1)	68.3% (±2.9)
combination	73.4% (±2.7)	77.5% (±2.6)

5 Conclusions and Future Research

In this article we investigated for the game of Havannah several enhancements in the play-out and selection step of MCTS. Based on the experimental results we offer three conclusions. The first conclusion we may draw is that by adding LGR and N-grams to the *play-out step* of MCTS a large performance gain is achieved, both when these two enhancements are used separately or in combination with each other. Especially, LGRF-2 and N-gram1 seem to be a strong combination.

The second conclusion we may give is that by using pattern knowledge to initialize the visit and win counts of the new nodes, the *selection step* is considerably enhanced. By biasing the selection towards joint/neighbor moves, local connections and edge/corner connections, a significant improvement in the playing strength of the MCTS program is observed.

For the third conclusion we may state that the best overall performance is achieved when visit-and-win-count initialization is combined with LGRF-2 and N-gram1. Experiments reveal a winning percentage of 77.5%.

There are several directions for future research. The first potential improvement is tweaking the parameter values for visit-and-win-count initialization. Furthermore, the combination of the three visit-and-win-count initialization heuristics may be improved. One may consider altering the importance of each of the three heuristics within the combination. A second idea is to let the importance of each of the three heuristics be dynamic with respect to the current stage of the game. For example, during the first stages of the game, one could bias the selection only towards joint and neighbor moves, because there are almost no chains yet on the board. As the game progresses and chains are formed, the importance of local and edge/corner connections may be increased while that of joint and neighbor moves is decreased.

Acknowledgments. We gratefully acknowledge earlier work on our Havannah-playing agent by Bart Joosten and Joscha-David Fossel as reported in their B.Sc. theses [11,8].

References

1. Arneson, B., Hayward, R.B., Henderson, P.: Monte Carlo Tree Search in Hex. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 251–258 (2010)
2. Baier, H., Drake, P.D.: The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 303–309 (2010)
3. Björnsson, Y., Finnsson, H.: CadiaPlayer: A Simulation-Based General Game Player. IEEE Transactions on Computational Intelligence and AI in Games 1(1), 4–15 (2009)
4. Chaslot, G.M.J.-B.: Monte-Carlo Tree Search. PhD thesis, Maastricht University, Maastricht, The Netherlands (2010)
5. 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)
6. 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)
7. Drake, P.D.: The Last-Good-Reply Policy for Monte-Carlo Go. ICGA Journal 32(4), 221–227 (2009)
8. Fossel, J.D.: Monte-Carlo Tree Search Applied to the Game of Havannah. Bachelor's thesis, Maastricht University, Maastricht, The Netherlands (2010)
9. Freeling, C.: Introducing Havannah. Abstract Games 14, 14–20 (2003)
10. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the 24th International Conference on Machine Learning, ICML 2007, pp. 273–280. ACM Press, New York (2007)
11. Joosten, B.: Creating a Havannah Playing Agent. Bachelor's thesis, Maastricht University, Maastricht, The Netherlands (2009)
12. Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)

13. 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)
14. Laramée, F.D.: Using N-Gram Statistical Models to Predict Player Behavior. In: Rabin, S. (ed.) AI Game Programming Wisdom, pp. 596–601. Charles River Media, Hingham (2002)
15. Lee, C.-S., Wang, M.-H., Chaslot, G.M.J.-B., 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 1(1), 73–89 (2009)
16. Lorentz, R.J.: Improving Monte–Carlo Tree Search in Havannah. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 105–115. Springer, Heidelberg (2011)
17. Nijssen, J(P.) A.M., Winands, M.H.M.: Enhancements for Multi-Player Monte-Carlo Tree Search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 238–249. Springer, Heidelberg (2011)
18. Rimmel, A., Teytaud, F.: Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcazar, A.I., Goh, C.-K., Merelo, J.J., Neri, F., Preuß, M., Togelius, J., Yannakakis, G.N. (eds.) EvoApplicatons 2010. LNCS, vol. 6024, pp. 201–210. Springer, Heidelberg (2010)
19. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo Simulations through RAVE Values. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 59–68. Springer, Heidelberg (2011)
20. Shannon, C.E.: Predication and Entropy of Printed English. The Bell System Technical Journal 30(1), 50–64 (1951)
21. Stankiewicz, J.A.: Knowledge-based Monte-Carlo Tree Search in Havannah. Master's thesis, Maastricht University, Maastricht, The Netherlands (2011)
22. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
23. Teytaud, F., Teytaud, O.: Creating an Upper-Confidence-Tree Program for Havannah. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 65–74. Springer, Heidelberg (2010)
24. Teytaud, F., Teytaud, O.: On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. In: Yannakakis, G.N., Togelius, J. (eds.) Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG 2010), pp. 359–364. IEEE Press (2010)
25. Winands, M.H.M., Björnsson, Y.: $\alpha\beta$-based Play-outs in Monte-Carlo Tree Search. In: 2011 IEEE Conference on Computational Intelligence and Games (CIG 2011), pp. 110–117. IEEE Press (2011)

Playout Search for Monte-Carlo Tree Search in Multi-player Games

J. (Pim) A.M. Nijssen and Mark H.M. Winands

Games and AI Group, Department of Knowledge Engineering,
Faculty of Humanities and Sciences,
Maastricht University, Maastricht, The Netherlands
{pim.nijssen,m.winands}@maastrichtuniversity.nl

Abstract. Over the past few years, Monte-Carlo Tree Search (MCTS) has become a popular search technique for playing multi-player games. In this paper we propose a technique called Playout Search. This enhancement allows the use of small searches in the playout phase of MCTS in order to improve the reliability of the playouts. We investigate max^{n}, Paranoid, and BRS for Playout Search and analyze their performance in two deterministic perfect-information multi-player games: Focus and Chinese Checkers. The experimental results show that Playout Search significantly increases the quality of the playouts in both games. However, it slows down the speed of the playouts, which outweighs the benefit of better playouts if the thinking time for the players is small. When the players are given a sufficient amount of thinking time, Playout Search employing Paranoid search is a significant improvement in the 4-player variant of Focus and the 3-player variant of Chinese Checkers.

1 Introduction

Deterministic perfect-information multi-player games pose an interesting challenge for computers. In the past the standard techniques to play these games were max^{n} [13] and Paranoid [20]. Similar to, for instance, Best Reply Search (BRS) [18] and Coalition-Mixer [12], these search techniques use an evaluation function to determine the values of the leaf nodes in the tree. Applying search is generally more difficult in multi-player games than in 2-player games. Pruning in the game tree of a multi-player game is much harder [19]. With $\alpha\beta$ pruning, the size of a tree in a 2-player game can be reduced from $O(b^d)$ to $O(b^{\frac{d}{2}})$ in the best case. In Paranoid, the size of the game tree can only be reduced to $O(b^{\frac{n-1}{n}d})$ in the best case and in BRS, the size can be reduced to $O\left((b(n-1))^{\lceil \frac{2d}{n} \rceil/2}\right)$. When using max^{n}, safe pruning is hardly possible. Also, opponent's moves are less predictable. Contrary to 2-player games, where two players always play against each other, in multi-player games (temporary) coalitions might occur. This can change the behavior of the opponents.

Over the past years, Monte-Carlo Tree Search (MCTS) [7,10] has become a popular technique for playing multi-player games. MCTS is a best-first search

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 72–83, 2012.
© Springer-Verlag Berlin Heidelberg 2012

technique that instead of an evaluation function uses simulations to guide the search. Next, MCTS is able to compute mixed equilibria in multi-player games [19], contrary to max^n, Paranoid and BRS. MCTS is used in a variety of multi-player games, such as Focus [15], Chinese Checkers [15,19], Hearts [19], Spades [19], and multi-player Go [5].

For MCTS, a tradeoff between search and knowledge has to be made. The more knowledge is added, the slower each playout gets. The trend seems to favor fast simulations with computationally light knowledge, although recently, adding more heuristic knowledge at the cost of slowing down the playouts has proven beneficial in some games [21]. Game-independent enhancements in the playout phase of MCTS such as Gibbs sampling [2] and RAVE [16] have proven to increase the playing strength of MCTS programs significantly. With ϵ-greedy playouts [19], some game-specific knowledge can be incorporated. Lorentz [11] improved the playing strength of the MCTS-based Havannah program WANDERER by checking whether the opponent has a 'mate-in-one' when selecting a move in the beginning of the playout. Winands and Björnsson [21] proposed $\alpha\beta$-based playouts for the 2-player game Lines of Action. Although computationally intensive, it significantly improved the playing strength of the MCTS program.

In this paper we propose Playout Search for MCTS in multi-player games. Instead of using computationally light knowledge in the playout phase, small two-ply searches are used to determine the moves to play. We test three different search techniques that may be used for Playout Search. These search techniques are max^n, Paranoid, and BRS. Playout Search is tested in two disparate multi-player games: Focus and Chinese Checkers.

The remainder of the paper is structured as follows. First, Section 2 gives a brief overview of the application of MCTS in multi-player games. Next, Playout Search is introduced in Section 3. An overview of the rules and domain knowledge for Focus and Chinese Checkers is given in Section 4. Subsequently, Section 5 describes the experiments and the results. Finally, the conclusions and an outlook on future research are given in Section 6.

2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [7,10] is a search technique that gradually builds up a search tree, guided by Monte-Carlo simulations. In contrast to classic search techniques such as $\alpha\beta$-search [9], it does not require a heuristic evaluation function. The MCTS algorithm consists of four phases [6]: selection, expansion, playout, and backpropagation (see Fig. 1). By repeating these four phases iteratively, the search tree is constructed gradually. Below we explain the application to multi-player games for our MCTS program [15].

In the *selection* phase the search tree is traversed from the root node until a node is found that contains children that have not been added to the tree yet. The tree is traversed using the Upper Confidence bounds applied to Trees (UCT) [10] selection strategy. In our program, we have enhanced UCT with Progressive History [15]. The child i with the highest score v_i in Formula 1 is selected.

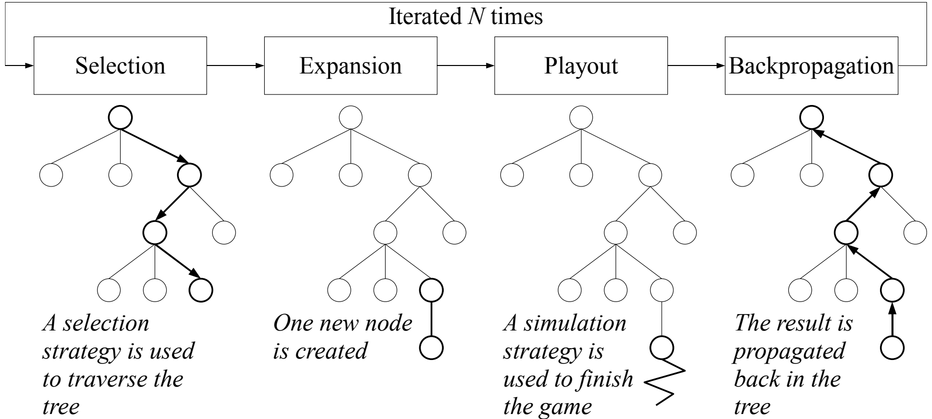


Fig. 1. Monte-Carlo Tree Search scheme (Slightly adapted from [6])

$$v_i = \frac{s_i}{n_i} + C \times \sqrt{\frac{\ln(n_p)}{n_i}} + \frac{s_a}{n_a} \times \frac{W}{n_i - s_i + 1} \tag{1}$$

In this formula, s_i denotes the total score of child i, where a win is being rewarded 1 point and a loss 0 points. The variables n_i and n_p denote the total number of times that child i and parent p have been visited, respectively. C is a constant, which determines the exploration factor of UCT. In the Progressive History part, s_a represents the score of move a, where each playout in which a was played resulted in a win adds 1 point and a loss 0 points. n_a is the number of times move a has been played in any previous playout. W is a constant that determines the influence of Progressive History.

In the *expansion* phase one node is added to the tree. Whenever a node is found which has children that have not been added to the tree yet, then one of these children is chosen and added to the tree [7].

During the *playout* phase, moves are played in self-play until the game is finished. Usually, the playouts are being generated using random move selection. However, progression has been identified as an important success factor for MCTS [8,22]. Ideally, each move should bring the game closer towards its conclusion. Otherwise, there is a risk of the simulations leading mostly to futile results. In slow-progressing games, such as Chinese Checkers and Focus (see Section 4), knowledge should be added to the playouts [3] to ensure a quick resolution of the game. Often, simple evaluations are used to select the moves to play. In our MCTS program, the following two strategies have been incorporated. (1) When using a *move evaluator*, a heuristic is used to assign a value to all valid moves of the current player. The move with the highest evaluation score is chosen. The move evaluator is fast, but it only considers a local area of the board. (2) With *one-ply search*, all valid moves of the current player are performed and the resulting board positions are evaluated. The move which gives the best board position, i.e., the highest evaluation score for the current player, is chosen.

The board evaluator is slower than the move evaluator, but it gives a more global evaluation. Knowledge can also be incorporated by employing 2-ply searches to determine the move to play. In Section 3 we explain which search techniques are used.

Finally, in the *backpropagation* phase, the result is propagated back along the previously traversed path up to the root node. In the multi-player variant of MCTS, the result is a tuple of size N, where N is the number of players. The value corresponding to the winning player is 1, the value corresponding to the other players is 0. The game-theoretic values of terminal nodes are stored and, if possible, backpropagated in such a way that MCTS is able to prove a (sub)tree [15,22].

This four-phase process is repeated either a fixed number of times, or until the time is up. When the process is finished, the child of the root node with the highest win rate is returned.

3 Playout Search

In this section we propose Playout Search for MCTS in multi-player games. In Subsection 3.1 we explain which search techniques are used in the playout phase. In Subsection 3.2 we describe which enhancements are used to speed up the search.

3.1 Search Techniques

Instead of playing random moves biased by computationally light knowledge in the playout phase, domain knowledge can be incorporated by performing small searches. This reduces the number of playouts per second significantly, but it improves the reliability of the playouts. When selecting a move in the playout phase, one of the following three search techniques is used to choose a move.

1) Two-ply max^n [13]. A two-ply $\max^{\mathrm{n}}$ search tree is built where the current player is the root player and the first opponent plays at the second ply. Both the root player and the first opponent try to maximize their own score. $\alpha\beta$-pruning in a two-ply $\max^{\mathrm{n}}$ search tree is not possible.

2) Two-ply Paranoid [20]. Similar to $\max^{\mathrm{n}}$, a two-ply search tree is built where the current player is the root player and the first opponent plays at the second ply. The root player tries to maximize its own score, while the first opponent tries to minimize the root player's score. In a two-ply Paranoid search tree, $\alpha\beta$-pruning is possible.

3) Two-ply Best Reply Search (BRS) [18]. BRS is similar to Paranoid search. The difference is that at the second ply, not only the moves of the first opponent are considered, but the moves of all opponents are investigated. Similar to Paranoid search, $\alpha\beta$-pruning is possible.

3.2 Search Enhancements

The major disadvantage of incorporating search in the playout phase of MCTS is the reduction of the number of playouts per second [21]. In order to prevent this reduction from outweighing the benefit of the quality of the playouts, enhancements should be implemented to speed up the search and keep the reduction of the number of playouts to a minimum. In our MCTS program, the following enhancements to speed up the playout search are used.

The number of searches can be reduced by using *ϵ-greedy playouts* [19]. With a probability of ϵ, a move is chosen uniform randomly. Otherwise, the selected search technique is used to select the best move. An additional advantage of ϵ-greedy playouts is that the presence of this random factor gives more varied playouts and prevents the playouts from being stuck in 'local optima', where all players keep moving back and forth. ϵ-greedy playouts are used with all aforementioned playout strategies.

The amount of $\alpha\beta$-pruning in a tree can be increased by using *move ordering*. When using move ordering, a player's moves are sorted using a static move evaluator. In the best case, the number of evaluated board positions in a two-ply search is reduced from b^2 to $2b - 1$ [9]. The size of the tree can be further reduced by using *k-best pruning*. Only the k best moves are investigated. This reduces the branching factor of the tree from b to k. The parameter k should be chosen such that it is significantly smaller than b, while avoiding the best move being pruned. Move ordering and k-best pruning are used in all techniques described in Subsection 3.1.

A second move ordering technique is applying *killer moves* [1]. In each search, two killer moves are always tried first. These are the two last moves that were best or caused a cutoff, at the current depth. Moreover, if the search is completed, the killer moves for that specific level in the playout are stored, such that they can be used during the next MCTS iterations. Killer moves are only used with search techniques where $\alpha\beta$-pruning is possible, i.e., Paranoid and BRS search.

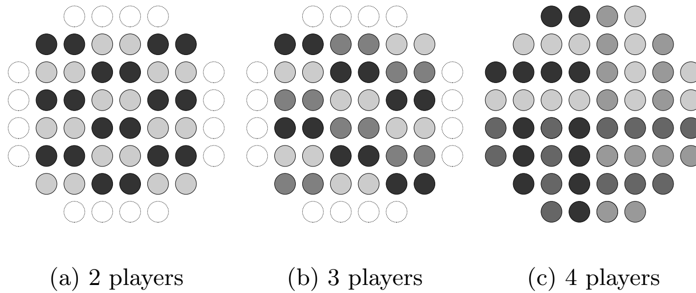
Other enhancements were tested, but they did not improve the performance of the MCTS program. The application of *transposition tables* [4] was tested, but the information gain did not compensate for the overhead. Also, *aspiration search* [14] did not speed up the search significantly. This can be attributed to the limited amount of pruning possible in a two-ply search tree.

4 Test Domains

Playout Search is tested in two different games: Focus and Chinese Checkers. In this section we briefly discuss the rules and the properties of Focus and Chinese Checkers in Subsection 4.1 and 4.2, respectively. In Subsection 4.3 we explain the move and board evaluators for Focus and Chinese Checkers.

4.1 Focus

Focus is an abstract multi-player strategy board game, which was invented in 1963 by Sid Sackson [17]. This game has also been released under the name



(a) 2 players (b) 3 players (c) 4 players

Fig. 2. Set-ups for Focus

Domination. Focus is played on an 8×8 board where in each corner three fields are removed. It can be played by 2, 3, or 4 players. Each player starts with a number of pieces on the board. In Fig. 2, the initial board positions for the 2-, 3-, and 4-player variants are given.

In Focus, pieces can be stacked on top of each other. A stack may contain up to 5 pieces. Each turn a player may move a stack orthogonally as many fields as the stack is tall. A player may only move a stack of pieces if a piece of his[1] color is on top of the stack. It is also allowed to split stacks into two smaller stacks. If a player decides to do so, then he only moves the upper stack as many fields as the number of pieces that are being moved.

If a stack lands on top of another stack, then the stacks are merged. If the merged stack has a size of $n > 5$, then the bottom $n - 5$ pieces are captured by the player, such that there are 5 pieces left. If a player captures one of his own pieces, he may later choose to place one piece back on the board, instead of moving a stack. This piece may be placed either on an empty field or on top of an existing stack.

There exist two variations of the game, each with a different winning condition. In the standard version of the game, a player has won if all other players cannot make a legal move. However, such games can take a long time to complete. Therefore, we chose to use the shortened version of the game. In this version, a player has won if he has either captured a certain number of pieces in total, or a number of pieces from each player. In the 2-player variant, a player wins if he has captured at least 6 pieces from the opponent. In the 3-player variant, a player has won if he has captured at least 3 pieces from both opponents or at least 10 pieces in total. In the 4-player variant, the goal is to capture at least 2 pieces from each opponent or to capture at least 10 pieces in total.

4.2 Chinese Checkers

Chinese Checkers is a board game that can be played by 2 to 6 players. This game was invented in 1893 and has since then been released by various publishers

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

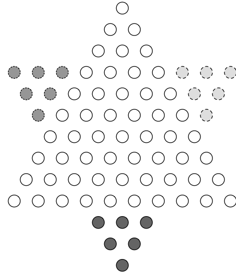


Fig. 3. A Chinese Checkers board [19]

under different names. Chinese Checkers is played on a star-shaped board. The most commonly used board contains 121 positions, where each player starts with 10 checkers. We decided to play on a slightly smaller board [19] (see Fig. 3). In this version, each player plays with 6 checkers. The advantage of a smaller board is that games take a shorter amount of time to complete, which means that more Monte-Carlo simulations can be performed and more experiments can be run. Also, it allows the use of a stronger evaluation function.

The goal of each player is to move all his pieces to his home base at the other side of the board. Pieces may move to one of the adjacent positions or they may jump over another piece to an empty position. It is also allowed to make multiple jumps with one piece in one turn, making it possible to create a setup that allows pieces to jump over a large distance. The first player who manages to fill his home base wins the game.

4.3 Domain Knowledge

For Chinese Checkers, the value of a move equals $d_s - d_t$, where d_s is the distance of the source location of the piece that is moved to the home base, and d_t the distance of the target location to the home base. For each location on the board, the distance to each home base is stored in a table. Note that the value of a move is negative if the piece moves away from the home base. For determining the board value, a lookup table [19] is used. This table stores, for each possible configuration of pieces, the minimum number of moves a player should perform to get all pieces in the home base, assuming that there are no opponents' pieces on the board. For any player, the value of a board equals $28 - m$, where m is the value stored in the table which corresponds to the configuration of the pieces of the player. Note that 28 is the highest value stored in the table.

For Focus, the value of a move equals $10(n + t) + s$, where n is the number of pieces moved, t is the number of pieces on the target location, and s is the number of stacks the player gained. The value of s can be 1, 0, or –1. For any player, the board value is based on the minimum number of pieces the player needs to capture to win the game, r, and the number of stacks the player controls, c. The score is calculated using the formula $600 - 100r + c$.

Table 1. 95% confidence intervals of some winning rates for 1500 games

Win percentage	Confidence interval
50%	± 2.5%
40% / 60%	± 2.5%
30% / 70%	± 2.3%
20% / 80%	± 2.0%

5 Experiments

In this section, we describe the experiments that were performed to investigate the strength of Playout Search for MCTS in Focus and Chinese Checkers. In Subsection 5.1 the experimental setup is given. In Subsection 5.2 we present the experimental results for the different search techniques of Playout Search in Focus and Chinese Checkers.

5.1 Experimental Setup

The MCTS engines of Focus and Chinese Checkers are written in Java [15]. For Formula 1, the constant C is set to 0.2 and W is set to 5. All players use ϵ-greedy playouts with $\epsilon = 0.05$. The value of k for k-best pruning is set to 5. These values were achieved by systematic testing. The experiments were run on a cluster containing of AMD64 Opteron 2.4 GHz processors. In order to test the performance of Playout Search, we performed several round-robin tournaments where each participating player uses a different playout strategy. These playout strategies include 2-ply $\max^n$ (M), 2-ply Paranoid (P), and 2-ply BRS (B). Additionally, we include players with one-ply (O) and move evaluator (E) playouts as reference players. The tournaments were run for 3-player and 4-player Chinese Checkers and 3-player and 4-player Focus. In each game, two different player types participate. If one player wins, a score of 1 is added to the total score of the corresponding player type. For both games, there may be an advantage regarding the order of play and the number of different players. In a 3-player game there are $2^3 = 8$ different player-type assignments. Games where only one player type is playing are not interesting, leaving 6 ways to assign player types. For four players, there are $2^4 - 2 = 14$ assignments. Each assignment is played multiple times until approximately 1,500 games are played and each assignment was played equally often. In Table 1, 95% confidence intervals of some winning rates for 1500 games are given.

5.2 Results

In the first set of experiments, all players were allowed to perform 5000 playouts per move. The results are given in Table 2. The numbers are the win percentages of the players denoted on the left against the players denoted at the top.

The results show that for 3-player Chinese Checkers, BRS is the best technique. It performs slightly better than $\max^n$ and Paranoid. BRS wins 53.4% of

Table 2. Round-robin tournament of the different search techniques in Chinese Checkers and Focus with 5000 playouts per move (win%)

	E	O	M	P	B	Avg.
Move eval.	-	25.2	20.9	21.2	18.3	21.4
One-ply	74.8	-	44.5	40.5	38.9	49.7
Max^n	79.1	55.5	-	48.1	46.6	57.3
Paranoid	78.8	59.5	51.9	-	49.1	59.8
BRS	81.7	61.1	53.4	50.9	-	61.8

3-player Chinese Checkers

	E	O	M	P	B	Avg.
Move eval.	-	44.5	38.4	38.5	33.3	38.7
One-ply	55.5	-	44.3	44.1	40.5	46.1
Max^n	61.6	55.7	-	52.0	45.2	53.6
Paranoid	61.5	55.9	48.0	-	44.5	52.5
BRS	66.7	59.5	54.8	55.5	-	59.1

3-player Focus

	E	O	M	P	B	Avg.
Move eval.	-	30.3	27.6	26.9	22.9	26.9
One-ply	69.7	-	47.4	45.1	39.7	50.5
Max^n	72.4	52.6	-	49.1	48.1	55.6
Paranoid	73.1	54.9	50.9	-	46.2	56.3
BRS	77.1	60.3	51.9	53.8	-	60.8

4-player Chinese Checkers

	E	O	M	P	B	Avg.
Move eval.	-	42.0	35.0	35.2	33.4	36.4
One-ply	58.0	-	43.3	42.6	40.1	46.0
Max^n	65.0	56.7	-	50.5	48.5	55.2
Paranoid	64.8	57.4	49.5	-	48.2	55.0
BRS	66.6	59.9	51.5	51.8	-	57.5

4-player Focus

Table 3. Playouts per second for each type of player in each game variant

Game	Move eval.	One-ply	Max^n	Paranoid	BRS
3-player Focus	7003	6138	2336	3356	1911
4-player Focus	6976	6237	2344	3410	1887
3-player Chinese Checkers	7322	6047	3439	4307	3890
4-player Chinese Checkers	5818	4630	2407	3066	2536

the games against max^n and 50.9% against Paranoid. These three techniques perform significantly better than one-ply and the move evaluator. The win rates against one-ply vary from 55.5% to 61.6% and against the move evaluator from 78.8% to 81.7%. In the 4-player variant, max^n, Paranoid and BRS remain the best techniques, where BRS performs slightly better than the other two. BRS wins 53.8% of the games against Paranoid and 51.9% against max^n. The win rates of max^n, Paranoid, and BRS vary from 72.4% to 77.1% against the move evaluator and from 52.6% to 60.3% against one-ply.

For 3-player Focus, the best technique is BRS, winning 54.8% against max^n and 55.5% against Paranoid. Max^n and Paranoid are equally strong. The win rates of max^n, Paranoid, and BRS vary between 61.5% and 66.7% against the move evaluator and between 55.7% and 59.5% against one-ply. BRS is also the best technique in 4-player Focus, though it is closely followed by max^n and Paranoid. BRS wins 51.5% of the games against max^n and 51.8% against Paranoid.

In the second set of experiments, we gave each player 5 seconds per move. For reference, Table 3 shows the average number of playouts per second for each type of player in each game variant. Note that at the start of the game, the number of playouts is smaller. As the game progresses, the playouts become shorter and the number of playouts per second increases.

Table 4. Round-robin tournament of the different search techniques in Chinese Checkers and Focus for time settings of 5 seconds per move (win%)

	E	O	M	P	B	Avg.
Move eval.	-	28.7	42.7	31.5	36.1	34.8
One-ply	71.3	-	62.5	50.8	58.2	60.7
Max^n	57.3	37.5	-	36.1	43.5	43.5
Paranoid	68.5	49.2	63.9	-	55.7	59.3
BRS	63.9	41.8	56.5	44.3	-	51.6

3-player Chinese Checkers

	E	O	M	P	B	Avg.
Move eval.	-	43.2	54.2	48.1	51.8	49.3
One-ply	56.8	-	58.9	53.9	57.9	56.9
Max^n	45.8	41.1	-	43.5	50.7	45.3
Paranoid	51.9	46.1	56.5	-	52.7	51.8
BRS	48.2	42.1	49.3	47.3	-	46.7

3-player Focus

	E	O	M	P	B	Avg.
Move eval.	-	33.7	45.9	35.4	42.9	39.5
One-ply	66.3	-	60.5	53.7	56.2	59.2
Max^n	54.1	39.5	-	40.3	46.6	45.1
Paranoid	64.6	46.3	59.7	-	56.2	56.7
BRS	57.1	43.8	53.4	43.8	-	49.5

4-player Chinese Checkers

	E	O	M	P	B	Avg.
Move eval.	-	42.3	41.1	40.1	43.9	49.1
One-ply	57.7	-	51.3	48.3	54.5	53.0
Max^n	58.9	48.7	-	47.9	55.9	52.9
Paranoid	59.9	51.7	52.1	-	54.3	54.5
BRS	56.1	45.5	44.1	45.7	-	47.9

4-player Focus

The results of the round-robin tournament are given in Table 4. In 3-player Chinese Checkers, one-ply and Paranoid are the best techniques. Paranoid wins 49.2% of the games against one-ply and 68.5% against the move evaluator. BRS ranks third, and the move evaluator and max^n are the weakest techniques. In 4-player Chinese Checkers, one-ply is the best technique, closely followed by Paranoid. One-ply wins 53.7% of the games against Paranoid. Paranoid is still stronger than the move evaluator, winning 64.6% of the games. BRS comes in third place, outperforming max^n and the move evaluator.

One-ply also performs the best in 3-player Focus. Paranoid plays slightly stronger than the move evaluator, with Paranoid winning 51.9% of the games against the move evaluator. One-ply wins 56.8% of the games against the move evaluator and 53.9% against Paranoid. The move evaluator and Paranoid perform better than BRS and max^n. In 4-player Focus, Paranoid performs better than in the 3-player version and outperforms one-ply. Paranoid wins 51.7% of the games against one-ply and 59.9% against the move evaluator. Max^n also performs significantly better than in the 3-player version. It is as strong as one-ply and better than the move evaluator, winning 58.9% of the games.

In the final set of experiments, we gave the players 30 seconds per move. Because these games take quite some time to complete, only the one-ply player and the Paranoid player were matched against each other. In the previous set of experiments, these two techniques turned out to be the strongest. The results are given in Table 5.

Paranoid appears to perform slightly better when the players receive 30 seconds per move compared to 5 seconds per move. In 3-player Chinese Checkers, Paranoid wins 53.9% of the games, compared to 49.2% with 5 seconds. In 4-player Chinese Checkers, 48.3% of the games are won by Paranoid, compared to 46.3% with 5 seconds. In 3-player Focus, the win rate of Paranoid increases from 46.1% with 5 seconds to 50.7% with 30 seconds and in 4-player Focus from 51.7% to 54.1%.

Table 5. Win rates of the Paranoid player against the one-ply player for time settings of 5 and 30 seconds per move

Game	5 seconds	30 seconds
3-player Chinese Checkers	49.2%	53.9%
4-player Chinese Checkers	46.3%	48.3%
3-player Focus	46.1%	50.7%
4-player Focus	51.7%	54.1%

6 Conclusions and Future Research

In this paper we proposed Playout Search for improving the playout phase of MCTS in multi-player games. We applied 2-ply max^n, Paranoid, and BRS searches to select the moves to play in the playout phase. Some enhancements, such as ϵ-greedy playouts, move ordering, killer moves, and k-best pruning were implemented to speed up the search.

The results show that Playout Search significantly improves the quality of the playouts in MCTS. This benefit is countered by a reduction of the number of playouts per second. Especially BRS and max^n suffer from this effect. Based on the experimental results we may conclude that Playout Search for multi-player games might be beneficial if the players receive sufficient thinking time and Paranoid search is employed. Under these conditions, Playout Search outperforms playouts using light heuristic knowledge in the 4-player variant of Focus and the 3-player variant of Chinese Checkers.

There are two directions for future research. First, it may be interesting to test Playout Search in other games as well. Second, the two-ply searches may be further optimized. Though a two-ply search will always be slower than a one-ply search, the current speed difference could be reduced further. This can be achieved, for instance, by improved move ordering or lazy evaluation functions.

References

1. Akl, S.G., Newborn, M.M.: The Principal Continuation and the Killer Heuristic. In: Proceedings of the ACM Annual Conference, pp. 466–473. ACM, New York (1977)
2. Björnsson, Y., Finnsson, H.: CadiaPlayer: A simulation-based general game player. IEEE Transactions on Computational Intelligence and AI in Games 1(1), 4–15 (2009)
3. Bouzy, B.: Associating domain-dependent knowledge and Monte Carlo approaches within a go program. Information Sciences 175(4), 247–257 (2005)
4. Breuker, D.M., Uiterwijk, J.W.H.H., van den Herik, H.J.: Replacement Schemes and Two-Level Tables. ICCA Journal 19(3), 175–180 (1996)
5. Cazenave, T.: Multi-player Go. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 50–59. Springer, Heidelberg (2008)
6. 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)

7. 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)
8. Finnsson, H., Björnsson, Y.: Simulation Control in General Game Playing Agents. In: IJCAI 2009 Workshop on General Intelligence in Game Playing Agents, pp. 21–26 (2009)
9. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)
10. 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)
11. Lorentz, R.J.: Improving Monte–Carlo Tree Search in Havannah. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 105–115. Springer, Heidelberg (2011)
12. Lorenz, U., Tscheuschner, T.: Player Modeling, Search Algorithms and Strategies in Multi-player Games. In: van den Herik, H.J., Hsu, S.-C., Hsu, T.-s., Donkers, H.H.L.M(J.) (eds.) ACG 11. LNCS, vol. 4250, pp. 210–224. Springer, Heidelberg (2006)
13. Luckhart, C., Irani, K.B.: An algorithmic solution of n-person games. In: Proceedings of the 5th National Conference on Artificial Intelligence (AAAI), vol. 1, pp. 158–162 (1986)
14. Marsland, T.A.: A review of game-tree pruning. ICCA Journal 9(1), 3–19 (1986)
15. Nijssen, J.A.M., Winands, M.H.M.: Enhancements for Multi-Player Monte-Carlo Tree Search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 238–249. Springer, Heidelberg (2011)
16. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo Simulations through RAVE Values. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 59–68. Springer, Heidelberg (2011)
17. Sackson, S.: A Gamut of Games. Random House, New York (1969)
18. Schadd, M.P.D., Winands, M.H.M.: Best Reply Search for Multiplayer Games. IEEE Transactions on Computational Intelligence and AI in Games 3(1), 57–66 (2011)
19. Sturtevant, N.R.: An Analysis of UCT in Multi-player Games. In: van den Herik, H.J., Xu, X., Ma, Z., Winands, M.H.M. (eds.) CG 2008. LNCS, vol. 5131, pp. 37–49. Springer, Heidelberg (2008)
20. Sturtevant, N.R., Korf, R.E.: On pruning techniques for multi-player games. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, pp. 201–207. AAAI Press / The MIT Press (2000)
21. Winands, M.H.M., Björnsson, Y.: $\alpha\beta$-based Play-outs in Monte-Carlo Tree Search. In: 2011 IEEE Conference on Computational Intelligence and Games (CIG 2011), pp. 110–117. IEEE Press (2011)
22. Winands, M.H.M., Björnsson, Y., Saito, J.-T.: Monte Carlo Tree Search in Lines of Action. IEEE Transactions on Computational Intelligence and AI in Games 2(4), 239–250 (2010)

Towards a Solution of 7x7 Go with Meta-MCTS

Cheng-Wei Chou[3], Ping-Chiang Chou, Hassen Doghmen[1], Chang-Shing Lee[2], Tsan-Cheng Su[3], Fabien Teytaud[1], Olivier Teytaud[1,2], Hui-Ming Wang[2], Mei-Hui Wang[2], Li-Wen Wu[2], and Shi-Jim Yen[3]

[1] TAO (Inria, Lri, Univ. Paris-Sud, UMR CNRS 8623), France
[2] Dept. of Computer Science and Information Engineering, National University of Tainan, Taiwan
[3] Dept. of Computer Science and Information Engineering, NDHU, Hualian, Taiwan

Abstract. Solving board games is a hard task, in particular for games in which classical tools such as alpha-beta and proof-number-search are somehow weak. In particular, Go is not solved (in any sense of solving, even the weakest) beyond 6x6. We here investigate the use of Meta-Monte-Carlo-Tree-Search, for building a huge 7x7 opening book. In particular, we report the twenty wins (out of twenty games) that were obtained recently in 7x7 Go against pros; we also show that in one of the games, with no human error, the pro might have won.

1 Introduction

The main approach for solving board games consists in using alpha-beta with several improvements such as transposition tables, killer moves, symmetry, and expert knowledge for ranking moves; proof-number-search[1] is used for preferring nodes with a small branching factor. Some successes have been obtained in particular for Ponnuki-Go [16] and there are results for all rectangular boards until 30 locations[17]. In the case of difficult games, these techniques have been combined with Monte-Carlo evaluation[13]. The use of Monte-Carlo techniques is often efficient in difficult board games and in particular for the game of Go[8]; it can be used for exact solving[5]; however, 7x7 remains beyond the capabilities of the current computers and algorithms.

Meta-MCTS has been proposed [2] (a mixing between nested Monte-Carlo[6] and Monte-Carlo Tree Search[8]) precisely for cases in which exact solutions are beyond our capabilities. Some variants have been applied in one-player cases[12]. This paper is devoted to (1) the application of Meta-MCTS for building huge opening books in 7x7 Go, and (2) to checking Meta-MCTS' ability to learn on specific variations. As in [2], we will see that human expertise is necessary for top-level performance; a main difference with 9x9, however, is that in 7x7 a reasonable computation time leads to openings outperforming a given finite set of variations - a difference in the context of this study (compared to previous works in 9x9) is that we work with an adjusted Komi, i.e., Komi 9.5 for White and Komi 8.5 for Black, so that the problem is well posed and a solution exists, at least if we trust the widely believed assumption that "7x7 fair Komi is 9". In all, the paper and the experiments are based on MoGoTW 4.86 Soissons.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 84–95, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Exact Komi in Go. In Go, two players play a game, leading to a partition of the board into a black area, a white area, and possible a no-man's land which does not belong to anyone (so-called Seki). Black has won if its area is bigger than White's area (usually modified by a bonus, termed Komi, which is intended to offset the advantage of Black who plays first). We here recall simple math around Komi, showing that there exists a Komi such that the game, in case of perfect play, is a draw.

Consider the 7x7 case. The board has size $S = 49$. Consider B the Black area (i.e., the number of black stones plus the number of empty locations surrounded by black stones, after removal of dead stones), and K the no-man's land, i.e., the Seki locations which do not belong to anyone. Then White score is $W = S - B - K$. The difference between White and Black territory is $(S - B - K) - B = S - K - 2B$. So the optimal strategy for Black consists in maximizing $2B + K$. If there is a number T such that B can ensure $2B + K = T$ (and no more than this), then the difference between the territories is, in case of optimal play, $S - T$; if the Komi is $S - T$ (this is an integer), it leads to a draw. The ideal Komi is $S - T$.

The above just shows that an ideal Komi exists, and that this Komi is an integer; it does not tell us which Komi is the good one. If there is no Seki at optimal play, then the Komi is odd in 7x7, because the board size is odd and $2B$ is certainly even. It is widely assumed that the ideal Komi is 9 and we will see that our results confirm this (but do not prove it). Section 2 contains technical points and Section 3 experimental results. Section 4 describes games against humans, Section 5 points to grid-based learning, and Section 6 gives our conclusion.

2 Technical Points

We present Monte-Carlo Tree Search in Subsection 2.1 and Meta-Monte-Carlo Tree Search in Subsection 2.2. Subsection 2.3 will present the score functions used in our Meta-MCTS algorithm, and Subsection 2.4 will discuss existing opening books for 7x7 Go.

2.1 Monte-Carlo Tree Search (MCTS)

Monte-Carlo Tree Search [8,10] is an algorithm for 1-player or 2-player games. A game is (here) a finite set of nodes, organized as a tree with a root. Each node n is of one of the following three types:

- max node (nodes in which the max player chooses the next state among descendants);
- min node (nodes in which the min player chooses the next state among descendants);
- terminal node; then, the node is equipped with a reward $Reward(n) \in [0, 1]$.

In all cases, we note $D(n)$ the set of children of node n. We assume, for the sake of simplicity, that the root node is a max node. We will consider algorithms which perform simulations; the first simulation is s_1, the second simulation is s_2, etc. Each simulation is a path in the game, from the root to a leaf. Each node n is equipped with the following four items.

- Possibly, some side information $I(n)$ ($I(n)$, and also $K_1(n)$ or $K_2(n)$ below, are meant as a short notation for the many MCTS improvements in the literature which are based on expert rules or patterns[8,7,11]).
- A father $F(n)$, which is the father node of n; this is not defined for the root.
- A value $V(n)$; this value is the Bellman value; it is known since [18,3] that $V(n)$, equal to the expected value of the reward if both players play optimally, is well defined.
- For each simulation index $t \in \{1, 2, 3, 4, \ldots\}$,
  - $n_t(n) \in \{0, 1, 2, \ldots\}$ is the number of simulations with index in $1, 2, \ldots, t-1$ including node n, possibly plus some constant $K_1(n)$:
$$n_t(n) = \#\{i < t; n \in s_i\} + K_1(n). \tag{1}$$
  - $w_t(n) \in \mathbb{R}$ is the sum of the rewards of the simulations with index in $1, 2, \ldots, t-1$ including node n, possibly plus some constant $K_2(n)$:
$$w_t(n) = \sum_{i<t; n \in s_i} reward(s_i) + K_2(n), \tag{2}$$
    where $reward(s_i)$ is $Reward(l)$ if l is the last node of simulation s_i.
  - If n is not the root, $score_t(n) = score(w_t(n), n_t(n), n_t(F(n)), I(n), t)$; we will see which properties we have, depending on the score function. The score function is usually an estimate of the quality of a node.

We consider algorithms as in Figure 1.

The algorithm relies on a good score formula. We refer to the many papers about MCTS for the scores classically used in MCTS implementations[8,7,11]. The score used for Meta-MCTS (following the same principles) will be discussed in Subsection 2.2.

2.2 Meta-MCTS

Meta-MCTS is MCTS in which the random policy is replaced by a MCTS policy. This makes simulations rather slow; but the tree of preferred moves is of high quality, and can efficiently be used as an opening book. The reader is referred to [6] for nested-Monte-Carlo (with good results, including world records, in some puzzles), and to [2] for nested MCTS, with application to Go. We might define extensions of Meta-MCTS as follows.

- MCTS is built on top of a default (somehow naive) playout policy.
- Meta-MCTS is built on top of a MCTS policy.
- Meta-Meta-MCTS is built on top of a Meta-MCTS policy.
- ...

To the best of our knowledge, such nested levels have been successfully used in Monte-Carlo, but not yet in MCTS; we here only use Meta-MCTS.

```
Input: a game.
for t = 1, 2, 3, ... (until no time left) do
   s_t ← ()     // empty simulation
   s = root(game) // the root of the game is the initial state
   while s is not terminal do
      s_t ← s_t.s    // s is added to the simulation
      switch s do
         case max node
            s ← arg max_{n∈D(s)} score(n)
         end
         case min node
            s ← arg min_{n∈D(s)} score(n)
         end
      end
      // the arg min or arg max can be replaced by a ε-greedy rule
      // or other stochastic rules
   end while
   s_t ← s_t.s    // s is added to the simulation
end for
decision = arg max_{n∈D(root)} n_t(n)      // decision rule
Output: a decision, i.e. the node in which to move.
```

Fig. 1. A framework of Monte-Carlo Tree Search. All usual implementations fall in this schema depending on the *score* function. The decision step is sometimes different, without impact on our results. The *score*(.) function, when the number of simulations is 0, defines the default policy (i.e., the policy when no statistics are available); when the score function does not depend on numbers of wins and on numbers of simulations, MCTS boils down to the old Monte-Carlo algorithms. We recall that $D(n)$ is the number of children of node n.

2.3 Scoring Functions Used in Our Meta-MCTS

In our Meta-MCTS, as well as in classical MCTS, we use a different rule for choosing moves when using our algorithm for really playing a game, than for choosing moves when using our algorithm for building the tree. The two rules are as follows:

1. play the move with highest empirical success rate. This rule is used for playing a real game;
2. play the move with highest empirical success rate, if such a move has success rate $\geq 10\%$; otherwise, choose a move by the default policy (in Meta-MCTS, the default policy is a MCTS). This rule is used in self-play games aimed at building the tree.

The reason for introducing the second rule is that the seemingly natural rule 1 is not consistent[4]. During the learning steps (i.e., for building the

opening book), it can concentrate on one move only, in spite of a success rate converging to 0, if all other moves have an empirical success rate 0 due to an unlucky first simulation. The typical example is as follows:

- there are two legal moves, one good (leading to a forced win) and one bad (leading to a loss);
- the good move has success rate 0, because it has been tested once, and, unluckily, it was a loss;
- the bad move has been tested N times, with only one win, and has therefore a success rate $1/N$.

The bad move will be tested again and again, because $1/N$ is always > 0, whereas the good move is never tested again and keeps a success rate 0. For more details on this anomaly, and in particular a consistency proof when using rule 2 during the tree building phase, we refer to [4]; alternate consistent rules consist in using a regularized score (number of wins plus K divided by number of simulations plus $2K$), or a Upper Confidence Bound-like formula - but using such a formula implies that all the nodes in the tree will be visited, whereas other rules above can lead to a smaller visited tree.

2.4 Existing Handcrafted 7x7 Openings

A classical partial solution for 7x7 has been proposed by Davies[9], and developed by several authors (posts on mailing lists refer to contributions by J. Tromp [14]). J. Tromp's homepage contains an interesting analysis [15]. However, the proposed opening is far from solving all cases.

Our methodology (detailed below) did not find any mistake in these openings. However, it developed interesting variations that were not, to the best of our knowledge, in these openings. Such a new variation is shown in Fig. 3 for Black; this variation was not analyzed after the third move in the existing openings. Also, many variants in professional games discussed below (Section 4) were not discussed in the existing openings; in particular, the critical variation D4-D5-E4-C4 leads to complicated variations and we have seen that even pros can make mistakes in it.

3 Experimental Results of Meta-MCTS Before Modifications by Human Expertise

We used Senseis' work (see homepage) on 7x7 Go in two manners: (1) as a preliminary opening book, and (2) as a sparring partner. This means that the Meta-MCTS tree is initialized at Senseis' solution, and is trained versus this solution. Meta-MCTS is based on a tree equipped with statistics, possibly encoded

as a set of games; this makes it easy and natural to include Senseis' work as a preliminary solution. For the use of Senseis' solution as a sparring partner, we proceeded as follows, e.g., for Black (the same, mutatis mutandis, for White):

- Black starts with a subset of the N Senseis variations.
- White is a MCTS algorithm possibly with opening book, and plays either
  - no opening at all (pure MCTS algorithm), with probability $1/(N+1)$;
  - one (randomly and uniformly chosen) variation of Senseis' opening, each of them with probability $1/(N+1)$.

Experimental results are provided in Fig. 2, Top (learning as Black) and Bottom (learning as White). The three main conclusions are as follows.

- Meta-MCTS quickly learned against any fixed set of variations: the curve increases, between each introduction of a new Senseis' variation. This shows

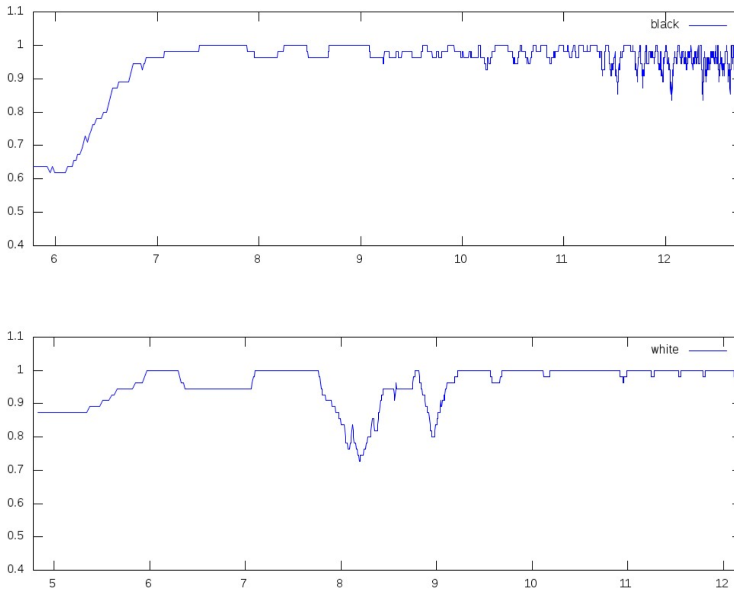


Fig. 2. These curves are learning curves of Meta-MCTS. Top: learning as Black (Komi 8.5). Bottom: learning as White (Komi 9.5). In both curves, the x-axis is log2(number of simulated games). Each game is of order 2h; so the total learning time (distributed on a cluster) is a few years of computation, distributed on Grid5000. The y-axis is the moving average (window of size 55) of the winning rate. The Komi is 9.5 when the learner is White and 8.5 when the learner is Black, so that if the right Komi is 9 then the learner can possibly reach 100%. At the beginning, Meta-MCTS learns against MoGoTW White with no opening; then variations of Senseis' SGF file are used as opening (they are randomly chosen, together with no opening at all); at this point the success rate starts to decrease, but it quickly increases again close to 100%.

that programs do not (by far) play perfectly without opening book; learning the opening book improves the success rate.
- Before the use of Senseis' openings, we might have believed that the program was playing almost perfectly as the success rate is close to 100%; however, the first games against Senseis' variations were not that good. After some learning, the program can play correctly against Senseis' variations and has developed many new variations.
- We will see below that even at this point, there were some important variations which were not yet analyzed; this confirms [9]'s claim that 7x7 Go is hard even at the best human level. More on this in later sections and in the conclusion.

4 Games against Humans

We tested the generated opening book against humans.

In order to check that the fair Komi is 9, we tested games as follows. MoGoTW plays as Black, with Komi 8.5; and MoGoTW plays as White, with Komi 9.5. If MoGoTW was unbeatable in this context, it would imply that the right Komi (in the sense: the Komi leading to a draw) is 9. We cannot know if a program is unbeatable just by a finite set of experiments, but we tested many games against many professional players. MoGoTW won all games; 10 games as Black against 10 different pros (Fig. 5), and 10 games as White against these same 10 pro players (Fig. 6). However, importantly, the second game won as Black by MoGoTW could have been a loss - MoGoTW made a mistake, but won thanks to a mistake by the human player; the corrected variation V, with the modification by Shi-Jim Yen and pros, is discussed in Fig. 4.

5 Introduction of Human Expertise in the Grid-Based Learning

Human experts found that a pro might have won a game: the second game as Black (see Fig. 4). Therefore, we reran the grid-based experiment (Meta-MCTS) specifically on this variation V - i.e. now the white opponent of the black Meta-MCTS is playing variation V. The complete run on the Grid is therefore as follows.

- **First set of experiments:** (already discussed in Section 3)
  - Meta-MCTS for Black with Komi 8.5; against White openings in Senseis' solution (introduced during the run, i.e., N is increasing).
  - Meta-MCTS for White with Komi 9.5; against Black openings in Senseis' solution (introduced during the run, i.e., N is increasing).

Basically, from the curves, Meta-MCTS was able to learn the correct solution against each Senseis variation, but sometimes it does not analyze a crucial variation - as the variation V discussed above.

– **Second set of experiments:** after the 20 games against pros, a variation V was found by a pro and corrected by Shi-Jim Yen (6D); we reran a Meta-MCTS for Black with Komi 8.5, against this specific variation. The Meta-MCTS quickly found good moves (see Fig. 3).

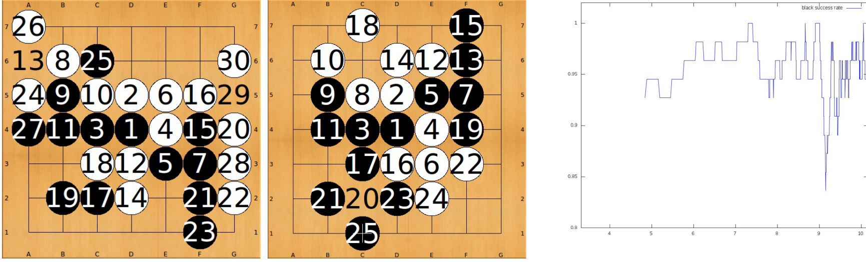


Fig. 3. Left: a bad move by Black (visible in games against variation V discussed in Section 4): black E3 should be black E5. Middle: the correction, as found by Meta-MCTS (after a long time) and by human Pro players - in this situation Black has a safe win. Right: the success rate of Meta-MCTS, when learning specifically on variation V.

All these experiments show that Meta-MCTS quickly adapts to given opponents: Fig. 2 shows that it could find good strategies against the MCTS algorithm, and against Senseis' variations; and Fig. 3 shows that it finds the solution to a new variation found by human experts. However, Meta-MCTS needs some external "coach", i.e., here, first, Senseis' variations, and, later, the variation V, for developing new knowledge in reasonable time.

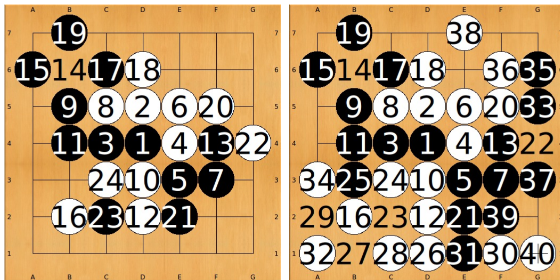


Fig. 4. Left: here an opening which was not correctly played by MoGoTW as Black; luckily for the bot, the human made a mistake and lost the game (see Fig. 5, second game, played by Chun-Yen Lin 2P). Right: the game that the human should have played in order to win.

Games played by Cheng-Jui Yu (3P), Chun-Yen Lin (2P), Hsiang-Chieh Wang (1P), Hsiang-Jen Huang (5P), Kai-Hsin Chang (4P), Shao-Chieh Ting (2P), Ting-Yi Lin (1P), Yin-Nan Chou (4P), Yu-Hsiang Lin (4P), and Yu-Pang Kou (1P) respectively.

Fig. 5. Games played as Black against pros

Games played by Cheng-Jui Yu (3P), Chun-Yen Lin (2P), Hsiang-Chieh Wang (1P), Hsiang-Jen Huang (5P), Kai-Hsin Chang (4P), Shao-Chieh Ting (2P), Ting-Yi Lin (1P), Yin-Nan Chou (4P), Yu-Hsiang Lin (4P), and Yu-Pang Kou (1P), respectively.

Fig. 6. Games played as White against pros

6 Conclusion

We applied directly the methodology from [2] to 7x7 Go. This methodology is Meta-MCTS (i.e., MCTS with a MCTS as a default policy), with archiving of statistics as an opening book. This provided a version of MoGoTW specialized for 7x7 Go, with a big opening book ($\tilde{1}2$,000 games). Many moves in games against pros were played automatically. Results in self-play were very good; yet, the games against the pros have shown that there was still a mistake; as the pro also made a mistake, we might have believed that MoGoTW had played perfectly - but a careful analysis by human experts has shown that humans had an opportunity of winning one of the games, and lost only because even pros can do mistakes in 7x7 Go. Incorporating variation V into the learning procedure provided good results: Meta-MCTS could find by itself the correction by Meta-MCTS. Nonetheless, if variation V had not been integrated in the experiments, Meta-MCTS might have not studied this variation (so, MoGoTW found the solution by itself, but it did not find the trouble by itself). Our main claims are summarized below.

– Meta-MCTS builds opening books which make MCTS stronger in 7x7 Go.
– MCTS plus an opening book learned by Meta-MCTS is much stronger than MCTS; however, there were still variations in Senseis' SGF file which were not correctly played by Meta-MCTS.
– MCTS plus Senseis' variations plus the opening book automatically learned by Meta-MCTS plus training against Senseis' variations is much stronger; however, our games against pros have shown that it was still possible for a pro to find a mistake in the opening. We have been lucky that the pro did not play correctly the rest of the game. This confirms that 7x7 can be quite hard even for the best humans.

Further Works. A possible further work is the exact solving of 7x7 Go. Our algorithm is not an exact solving algorithm; but it provides rather strong opening books. Maybe a possible approach is the following:

– make MoGoTW deterministic (just by using a fixed random seed);
– play all possible openings against MoGoTW, collect the leafs of the opening book;
– solve all these leafs.

A different further work would be the experimentation of Meta-MCTS in real-time. We all know that MCTS is weak in, e.g., Semeai and in combining several local fights; maybe Meta-MCTS can be a successful new approach, in the mid-game and not only in the opening.

Acknowledgements. We are grateful to Grid5000 for support in parallel versions of the algorithms, to the BIRS seminar on Combinatorial Game Theory, to the Dagstuhl seminar on the Theory of Evolutionary Algorithms, and to the Bielefeld seminar on Search Methods, to National Science Council (Taiwan) for NSC grants 99-2923-E-024-003-MY3 and NSC 100-2811-E-024-001.

References

1. Allis, L.V.: Searching for solutions in games and artificial intelligence. Ph.D. dissertation, The Netherlands: University of Limburg (1994)
2. 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: Giacobini, M., Brabazon, A., Cagnoni, S., Di Caro, G.A., Ekárt, A., Esparcia-Alcázar, A.I., Farooq, M., Fink, A., Machado, P. (eds.) EvoWorkshops 2009. LNCS, vol. 5484, pp. 323–332. Springer, Heidelberg (2009)
3. Bellman, R.: Dynamic Programming. Princeton Univ. Press (1957)
4. Berthier, V., Doghmen, H., Teytaud, O.: Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. In: Blum, C., Battiti, R. (eds.) LION 4. LNCS, vol. 6073, pp. 111–124. Springer, Heidelberg (2010)
5. Cazenave, T., Saffidine, A.: Score Bounded Monte-Carlo Tree Search. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 93–104. Springer, Heidelberg (2011)
6. Cazenave, T.: Nested monte-carlo search. In: IJCAI, pp. 456–461 (2009)
7. Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., 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), papers\pMCTS.pdf
8. 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)
9. Davies, J.: 7x7 go. American GO Journal 29(3), 11 (1995)
10. 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)
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), http://hal.inria.fr/inria-00369786/en/
12. Méhat, J., Cazenave, T.: Combining uct and nested monte carlo search for single-player general game playing. IEEE Trans. Comput. Intellig. and AI in Games 2(4), 271–277 (2010)
13. Saito, J.-T., Chaslot, G., Uiterwijk, J.W.H.M., Van Den Herik, H.J.: Monte-Carlo Proof-Number Search for Computer Go. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 50–61. Springer, Heidelberg (2007)
14. Sensei website, http://senseis.xmp.net/?7x7BestPlay
15. Tromp website, http://www.cwi.nl/~tromp/java/go/7x7.sgf
16. van der Werf, E.C.D., Uiterwijk, J.W.H.M., van den Herik, H.J.: Solving ponnuki-go on small boards. In: GAME-ON (2002)
17. van der Werf, E.C.D., Winands, M.H.M.: Solving go for rectangular boards. ICGA Journal 32(2), 77–88 (2009)
18. Zermelo, E.: Uber eine anwendung der mengenlehre auf die theorie des schachspiels. In: Proc. Fifth Congress Mathematicians, pp. 501–504. Cambridge University Press

MCTS Experiments on the Voronoi Game

Bruno Bouzy, Marc Métivier, and Damien Pellier

LIPADE, Université Paris Descartes, France
{bruno.bouzy,marc.metivier,damien.pellier}@parisdescartes.fr

Abstract. Monte-Carlo Tree Search (MCTS) is a powerful tool in games with a finite branching factor. The paper describes an artificial player playing the Voronoi game, a game with an infinite branching factor. First, it shows how to use MCTS on a discretization of the Voronoi game, and the effects of enhancements such as RAVE and Gaussian processes (GP). Then a set of experimental results shows that MCTS with UCB+RAVE or with UCB+GP are good first solutions for playing the Voronoi game without domain-dependent knowledge. Moreover, the paper shows how the playing level can be greatly improved by using geometrical knowledge about Voronoi diagrams. The balance of diagrams is the key concept. A new set of experimental results shows that a player using MCTS and geometrical knowledge outperforms a player without knowledge.

1 Introduction

UCT [20], Monte-Carlo Tree Search (MCTS) [12,8] and RAVE [17] are quite powerful tools in computer games for games with a finite branching factor. Voronoi diagrams are classical tools in image processing [4,25]. They have been used to define the Voronoi game (VG) [24,14], a game with an infinite branching factor. The VG is a good test-bed for MCTS and its enhancements. Furthermore, Gaussian processes (GP) are adapted to find the optimum of a target function in domains with an infinite set of states or actions [23,6]. Combining MCTS techniques with GP, and testing the result on the VG is the first goal of this paper. Our first set of results shows that MCTS with RAVE and GP leads to an effective solution. In the second stage (section 6), this paper shows that domain-dependent knowledge concerning Voronoi diagrams cannot be omitted. The balance of the diagrams is a key concept. The paper shows how to design an MCTS VG player that focuses on the balance of cells. Knowledge is used a priori to select a subset of interesting moves used in the tree and in the simulations. The second set of results shows that the MCTS player using Voronoi knowledge outperforms MCTS without knowledge.

The outline of the paper is the following. Section 2 defines the Voronoi game. Section 3 mentions work about MCTS, UCT, UCB, and RAVE. Section 4 explains how to combine GP and UCB within an MCTS program. Section 5 presents the results obtained by MCTS and GP. Before arriving at a conclusion, section 6 presents the insertion of relevant Voronoi knowledge into the MCTS algorithm to improve its playing level quite significantly. Section 7 gives our conclusions.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 96–107, 2012.
© Springer-Verlag Berlin Heidelberg 2012

2 Voronoi Game

The Voronoi game can be played by several players represented with a color [14]. At his[1] turn, a player puts a site of his color on a 2D square. Each cell around a site obtains the color of the site. A colored cell has an area. The area of a player is the sum of the areas of his cells. The game lasts a fixed number of turns (10 in the current work). At the end, the player who possesses the largest area wins. In its basic version and in the paper, there are two players, Red and Blue [24], the square is $C = [0, 1] \times [0, 1]$, and Red starts. Figure 1 shows the positions of a VG of length 10. Since Blue plays after Red, a *komi* can be introduced such that Red wins if the area evaluation, Red area minus Blue area, is superior to the *komi*, the *komi* value being negative.

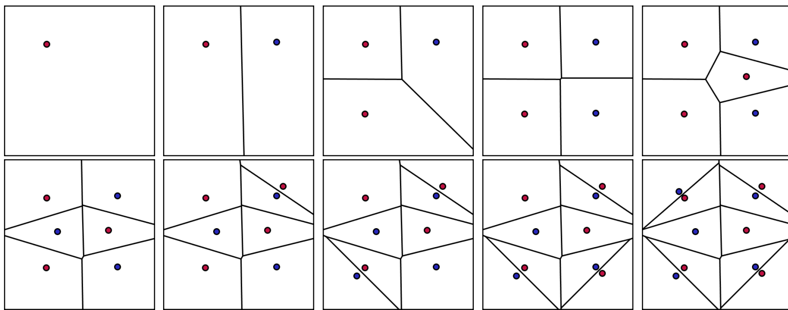


Fig. 1. The positions of the Voronoi game of length 10

Several methods can be used to compute Voronoi diagrams [4,18,16,25,22]. In order to go forward or backward in a game sequence, the incremental version [4] is appropriate. The computation of areas was helped by [13].

VG can be played on a circle, a segment, a rectangle, or a n-dimensional space as well. VG can be played in an N-round version (the players moves alternately one site per move) or in a one-round version (the players move all their sites in one move) [10]. Key points are essential to play the VG well [1]. Concerning the one-round VG played on a square, [15] proves that the second player always wins. Currently, Jens Anuth's master thesis is the most advanced and useful work about VG [2].

3 MCTS, UCT, UCB, and RAVE

MCTS (or UCT) repetitively launches simulations starting from the current state. It has four steps: the selection step, the expansion step, the simulation step, and the updating step. In the selection step, MCTS browses a tree from the root down to a leaf by using the UCB rule. To select the next state, UCT chooses the child maximizing the sum of two terms, the mean value and a UCB

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

term. If $x_1, \ldots, x_k$ are the k children of the current state, then the next state x_{next} is selected as follows:

$$x_{next} = \underset{x \in \{x_1, \ldots, x_k\}}{\arg\max} (\mu(x) + ucb(x)) \tag{1}$$

$$ucb(x_i) = \sqrt{C \times \frac{log(T)}{N_i}} \tag{2}$$

$\mu(x_i)$ is the mean value of returns observed. T is the total number of simulations in the current state. N_i is the number of simulations in state x_i. C is a constant value set up with experiments. When MCTS reaches a leaf of the tree, it creates one node. Then the simulation uses a random-based policy until the end of the game. At the final state, it receives the return and updates all the mean values of the states encountered in the tree. The UCB rule above balances between exploiting (choose the move with a good mean value to refine existing knowledge) or exploring (choose a move with few trials). The UCB rule works well when the set of moves is finite and pre-defined.

Furthermore, for the game of Go, the RAVE heuristic gave good results [17]. RAVE computes two mean values: the usual one and the AMAF (All Moves As First). The AMAF mean value is updated by considering that all moves of the same color of the first move of the simulation could have been played as the first move. Therefore, after a simulation, it is possible to update the AMAF mean value of all these moves with the return. RAVE uses a weighted mean value of these two mean values. β is the weight driving the balance between the AMAF mean value and the true mean value. K is a parameter set up experimentally. When a few simulations are performed, $\beta \approx 0$. When many simulations are performed, $\beta \approx 1$, the weight is on the usual mean value.

$$m_{RAVE} = \beta \times m + (1 - \beta) \times m_{AMAF} \tag{3}$$

$$\beta = \sqrt{\frac{N_i}{K + N_i}} \tag{4}$$

4 Light Gaussian Processes

Work concerning bandits on an infinite set of actions is facing the regret of not playing the best action [19,3]. The issue is how to generate a finite set of moves that can be provided to UCB to select a move. Gaussian processes (GP) [23,6] answer the question. At time t, the target function has been observed on points x_i with $i = 1, \ldots, N$. GP aims at finding the next point to try at time $t + 1$, either a new point or an observed point. The target function is approximated by a surrogate function called f. GP use an acquisition function A to choose the next point at time $t + 1$. The GP selection needs a matrix inversion which costs a large amount of CPU. Since GP selection is called at each node browsed by

the tree part of the simulations, it must be fast to execute. Therefore, we defined a specific algorithm lighter than GP.

The acquisition function A is defined as a sum of two functions f and g:

$$\forall x \in C,\ A(x) = f(x) + g(x) \tag{5}$$

For observed points, functions f and g are defined as follows:

$$\forall x \in \{x_1, \ldots, x_N\},\ f(x) = \mu(x) \tag{6a}$$

$$g(x) = ucb(x) \tag{6b}$$

For never observed points, function f is defined as a weighted mean over the mean values of the observed points:

$$\forall x \notin \{x_1, \ldots, x_N\},\ f(x) = \frac{\sum_{i=1}^{N} \mu(x_i) \times \exp(-a(x - x_i)^2)}{\sum_{i=1}^{N} \exp(-a(x - x_i)^2)} \tag{7}$$

where a is a constant. Function g is defined as follows:

$$\forall x \notin \{x_1, \ldots, x_N\},\ g(x) = G \times \prod_{i=1}^{N} (1 - \exp(-b(x - x_i)^2)) \tag{8}$$

where G and b are constant values.

Finally, the next point to be observed is selected according to the following rule:

$$x_{t+1} = \arg\max_{x \in D} A(x) \tag{9}$$

where D is a discretization in advance, or finite subset of C sufficiently large for accuracy, and sufficiently small to evaluate all its elements in practice at every time step. When browsing the UCT tree from the root node to a leaf node, the maximization process above is performed in each node. When a leaf node is expanded at the end of browsing, all the children are created following the discretization D. When a node is created, it is virtual, or not observed. It remains virtual until it is tried once, in which case it becomes observed. The size of D is crucial for the speed of the algorithm. When the process terminates, the algorithm returns the move that has the most trials.

The acquisition function A is not continuous: for an observed point x_i, $A(x_i)$ is superior to $A(x)$ for x in a small neighborhood of x_i. This allows two kinds of exploration. When x_{t+1} is an observed point, the exploration corresponds to a better estimation of $\mu(x_{t+1})$, or UCB exploration. Otherwise, the exploration corresponds to the observation of a new point in D. The competition between the two explorations avoids to explore new points before having sufficiently precise estimations of mean values of observed points. The dilemma between the two explorations is managed by G and C.

5 MCTS Experiments without Knowledge

In this section, the calibration experiments show the relevance of UCT (in 5.1) and UCT, RAVE, and GP (in 5.2) to play without domain-dependent knowledge.

5.1 Calibration Experiments

Square C is replaced by a discretization D containing $discret \times discret$ points. *discret* depends on each player. Its value is tuned thanks to calibration tournaments between several instances of the same player with different values of *discret*.

Without *komi*, Blue wins about 85% of the games on average, which hinders the search of the best players in a given set. For speeding this search, all games are launched with a *komi*. The value of the komi depends on the set of players considered. A satisfying value is determined experimentally to make Red and Blue win about 50% of games on average. Experimentally, we use a $komi \in [-0.04, -0.08]$. The value is negative reflecting that Blue has the advantage of playing the actually important last move.

All experiments are launched with an appropriate simulation number Ns. We used $Ns = 4000$. With such a value, a player spends between 5 and 10 minutes thinking time, and a game lasts between 10 and 20 minutes (a player uses between one and two minutes per move, about 80 random games per second). To compare two players A and B, we launch 50 games with A playing Red and 50 games with A playing Blue. 100 games enables the results to obtain $\sigma = 5\%$.

The values of Ns and *discret* interact. For low values of *discret*, few moves are considered. Although they can be sampled many times, this results in a poor level of play. For large values of *discret* many moves are considered, but they cannot be sampled sufficiently, resulting in a poor level of play as well. For intermediate values, the resulting program plays at its optimal level. Ns being set to 4000, each player has its best value of *discret*. For UCT, we experimentally found $discret = 20$.

Tuning C, the UCT constant, is mandatory to make UCT play well. Our experiments showed that $C = 0.25$ is a good value.

5.2 UCT, RAVE, and Light GP Experiments

Table 1 contains the results of an all-against-all tournament between UCT, RAVE, GP, and RGP using $Ns = 4000$. RAVE uses $discret = 16$ and $K = 400$. GP uses $discret = 26$, $G = 2$, and $a = b = 180$. RGP uses both enhancements: RAVE+GP. RGP uses $discret = 26$, $G = 2$, and $a = b = 180$. All these values were set in advance, by the calibration experiments.

First, RAVE is superior to UCT ($60.5\% \pm 3\%$) and GP is superior to UCT ($60\% \pm 3\%$). Second, RGP is slightly superior to RAVE ($56\% \pm 3\%$) showing that GP is a small enhancement when RAVE is on. RGP is neither inferior nor superior to GP ($51\% \pm 3\%$) showing that RAVE is not an efficient enhancement when GP is on. Third, GP is superior to RAVE ($55\% \pm 3\%$), which would show

Table 1. All-against-all results. The cell of row R and column C contains the result of 100 games between R playing Red and C playing Blue: the first number is the number of wins of Red. T indicates the total number of wins. $komi = -0.08$. $Ns = 4000$.

	UCT	*RAVE*	*GP*	*RGP*	T
UCT	39-61	30-70	43-57	46-54	350
RAVE	51-49	56-44	53-47	51-49	399
GP	63-37	63-37	78-22	75-25	428
RGP	55-45	63-37	77-23	80-20	423

that GP is a better enhancement than RAVE. Fourth, the bad news is that RGP is slightly superior only to UCT ($54.5\% \pm 5\%$), which shows that enhancements are significant by themselves but that their sum is not. Fifth, overall, GP is the best player of our set of experiments regarding the all-against-all tournament total win number. However, time considerations must be underlined: GP and RGP spend 13 minutes per game per player, while UCT and RAVE spend 5 minutes per game per player. GP (even in its light version) have a heavy computational cost. A fairer assessment between players would give the same thinking time to all players. This would negate the positive effect of GP.

6 MCTS Experiments with Knowledge

This section shows how to insert Voronoi knowledge into the MCTS framework above, and underlines the improvement in terms of playing level. Jens Anuth's thesis describing smart evaluation functions including Voronoi knowledge [2] motivated the work presented in this section. The outline below follows the strategical structure of the game (in 6.1 to 6.7). In 6.8 we consider human players and other work.

- the last move special case,
- simple attacks on unbalanced cells and the biggest cell attack,
- balance: balanced cells, DB: a defensive balanced player,
- BUCT: a UCT player using balanced VD,
- aBUCT: a BUCT player using simple attacks in the simulations,
- the one-round game and a second-player strategy,
- ABUCT: a BUCT player using sophisticated attacks in the tree.

6.1 The Last Move Special Case

The last move is a special case: since the other player will not play any move after the last move, a depth-one search is the appropriate tool. The result only depends on the discretization of the square. The higher the discretization, the better the optimum. The before-last move is also a special case: if the computing power is sufficient, performing a depth-two search at the before-last move offers the same upsides than depth-one search at the last move. In the following subsections, all the players described are assessed by assuming that the last and before-last moves are played with depth-one or depth-two minimax strategy, respectively.

6.2 Simple Attacks on Unbalanced Cells and the Biggest Cell Attack

A cell has a gravity center. A straight line intersecting the gravity center of a cell splits this cell into two parts of which the size is the half of the given cell. A cell of which the site is situated on its gravity center is called a balanced cell, an unbalanced cell otherwise. Figure 2 shows a game that illustrates balanced and unbalanced cells. After moves 1 and 2, the two cells are clearly unbalanced. After move 3, the cells are almost balanced. The most basic attack on an unbalanced cell consists in occupying its gravity center, and consequently in stealing more than the half of the cell. Moves 7 and 8 are straightforward attacks succeeding on clearly unbalanced cells. Computing the gravity center of a cell is fast and performed simultaneously with the area computation. A straightforward and fast player can be designed by choosing the biggest cell of the opponent and playing on its gravity center. We call such a player the biggest-cell-attack player (BCA). The BCA player is quite effective against any player creating unbalanced cells without special purpose. Particularly, the BCA player itself produces rather unbalanced cells. BCA offers to its opponent the same weakness as the weakness he is exploiting. See moves 5 and 6 of figure 2. Furthermore, the BCA player remains inefficient against players producing balanced cells. Moves 5 and 6 are straightforward attacks on two cells almost balanced. It is worth noting that BCA wins 60% of the games against 400-point-depth-one search. For this reason, in the following, we added the moves generated by the BCA strategy into the set of moves used by depth-one search.

6.3 Balance

Since the BCA player is dangerous for unbalanced cells, playing cells as balanced as possible becomes crucial.

Balanced cells. In the VG played on a circle, Ahn defines the importance of key points [1]. The location of a key point does not depend on the player. In a

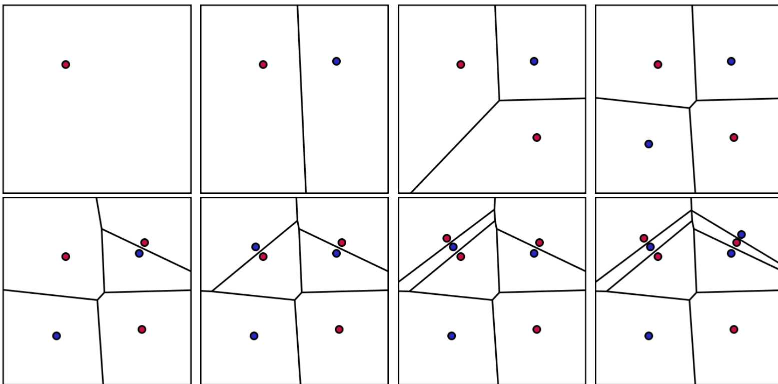


Fig. 2. A Voronoi game with straightforward cell attacks

$2 \times N$ round VG, there are N key points that are equally distributed in the circle. Playing on them is advised by the best strategy [1]. On the square [10,15,2] or any polygon [2], key points are strategically important too. Determining the N key points can be performed by the Lloyd algorithm [21]. The Lloyd algorithm is iterative. It starts with N sites randomly chosen in the square. At each iteration, it moves every sites to the gravity centers of the cells, and computes the resulting VD. In our work, the algorithm stops when the biggest distance between a site and a gravity center reaches a threshold. Since the number of iterations is finite, the VD output of the Lloyd algorithm is not exactly balanced. Figure 3 shows three examples of one-color balanced diagrams for the square.

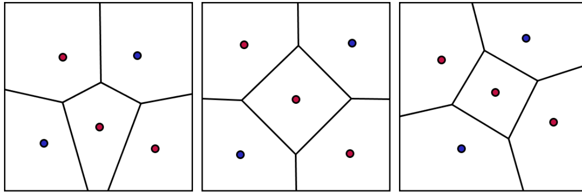


Fig. 3. Three one-color balanced diagrams with 5 sites for ten-site Voronoi Game

DB: A Defensive Balanced Player A staightforward defensive program can easily be designed: the defensive and balanced player (DB). In advance, DB computes a balanced one-color VD, and follows it blindly by picking one site at each turn. DB including the last move special case is quite a good player. If the balanced diagrams are computed offline, DB plays every move instantly. The result is impressive: DB vs $UCT(ns = 1000)$: 95% and DB vs $UCT(ns = 4000)$: 70%.

6.4 BUCT: A UCT Player Following Balanced VD in the Simulations

Since balance is important, we aim at integrating it into a UCT player that follows balanced VD in the simulations. Let us call BUCT such a player.

Guessing the Best Balanced One-Color VD Compatible with the Past Moves. Past moves of the actual game cannot be moved and they have no reason to give balanced one-color diagrams. Therefore, for each color, BUCT simulations must follow as almost balanced one-color VD compatible with the past moves. Here, the Lloyd algorithm must work with some unmovable sites. This raises two obstacles. First, the output diagram of the Lloyd algorithm is not necessarily balanced anymore. Second, some random initializations of the opponent sites lead the Lloyd algorithm to local optima. Therefore, the Lloyd algorithm must be launched several times to avoid local optima, and, for each color, the best balanced diagram is kept.

Using the Best Balanced One-Color VD. During the simulations the opponent moves are slight variations around the opponent balanced one-color VD,

and the friendly moves are slight variations around the friendly balanced one color VD. BUCT is improved by the last move special case as well. The results are rather encouraging against UCT: with 1000 simulations BUCT wins 98% of games, 90% (2000 simulations) and 85% (4000 simulations). However, BUCT with 1000, 2000 or 4000 simulations only wins 25% against DB. At this point, the ranking is: $UCT \ll BUCT \ll DB$. UCT has been enhanced into BUCT, but BUCT remains inferior to DB.

6.5 aBUCT: A BUCT Player Using Simple Attacks in the Simulations

Since the BCA strategy is efficient and fast to compute, we have added the BCA strategy in the simulations, which gives a player named aBUCT. In the simulations, the BCA strategy is drawn with probability A^{1-L} where L is the number of remaining moves in the simulation. The lower the move number in the simulation, the lower the probability to play a BCA move. If the BCA strategy is not drawn then the moves are generated according to the simulation policy of BUCT. aBUCT performs quite well: with 500 simulations only, it wins 55% of games against DB, and becomes the best player at this point. This result was obtained with $A = 2$. Other A values in $[1, 4]$ were tested but gave worse results. We observe that adding simple attacks in the simulations improve BUCT from 20% up to 55% against DB, which means BCA is a quite efficient enhancement. However, we saw that BCA has difficulties against balanced diagrams. Therefore more sophisticated attacks should give better results. At this point we have: $UCT \ll DB \leq aBUCT$.

6.6 The One-Round Game and the Second Player Strategy 1R2P

Since the one-color diagram balance is important, let us consider the one-round game [10]. The one-round VG differs from the N-round VG in that Red plays all his sites first (in one round), and Blue plays all his sites second. We are interested in finding out which second-player strategies defeat arbitrary red VD. Assume a red diagram is given with N cells. Blue, the second player, has N moves to play in a row to win the game. This is a planning problem. To solve it, a straightforward tool consists in playing the BCA strategy N times. This tool works well for any unbalanced red diagram. Then, a smarter tool is to use the idea of Fekete [15]. It consists in playing the first move by a depth-one search, and the following moves by the BCA strategy. It works on the square example and $N = 4$ in [15]. One may extend Fekete's idea by using the one-round second player (1R2P) strategy.

```
B=0
repeat
    play the depth-one strategy B times
    play the BCA strategy N-B times
    B=B+1
until success or B>N
```

The 1R2P strategy iteratively tries combinations of playing depth-one strategies B times and BCA strategies $N - B$ times. While failures are encountered, 1R2P tries to solve the problem again by incrementing B. Theoretically, since complex combinations of blue sites might be necessary to beat complex red balanced diagrams, the algorithm 1R2P is not proven to be complete. However, in practice, along all our experiments, 1R2P always returned a successful strategy. With $N = 5$, 1R2P never needed B being greater than 2. Let $Aggressive(V)$ be the blue diagram obtained by 1R2P in the one-round VG starting with the red diagram V.

6.7 ABUCT: A BUCT Player Using Sophisticated Attacks in the Tree

With the possibility to build aggressive diagrams defeating balanced diagrams, we are now able to set up a new player called Agressive and Balanced UCT (ABUCT). In advance, ABUCT computes the red and blue balanced diagrams adequately to the past moves actually played: $RedBD$ and $BlueBD$. Then, ABUCT computes $Aggressive(RedBD)$ and $Aggressive(BlueBD)$ with the 1R2P strategy on the one-round game. Then, ABUCT launches the UCT simulations by moves generated according to the balanced diagrams and to the agressive diagrams in the tree part of UCT. The results are excellent. With 500 simulations only, ABUCT wins 71% against aBUCT, 93% against UCT (500 simulations as well), and importantly 77% against DB. To sum up, we have: $UCT \ll DB \leq aBUCT \ll ABUCT$.

6.8 Against Human Players and against Other Work

As seen above, for adequate play, playing diagrams as balanced as possible is crucial. Human players (H) can be good in roughly seeing the balance of a diagram. However, they cannot be as precise as the Lloyd algorithm. Furthermore, the precision of a mouse click on a GUI point is a real burden for human players. This lack of precision limits the human level below the level of plain artificial players such as BCA. Despite of this handicap, human players may defeat UCT of section 3. Finally, we would like to remark that other work exists in some applets [24,14] using simple players such as BCA, and the work by Jens Anuth [2]. Although Anuth's program can play on arbitrary polygons, the best player of Anuth's work corresponds to DB. We observed that: $UCT \leq H \leq BCA \ll DB \ll ABUCT$.

7 Conclusion

We have shown a successful adaptation of MCTS in a game with an infinite branching factor, the Voronoi game. We tested UCT with RAVE and GP. They gave results strictly better than UCT alone (60% of wins). Adding Voronoi knowledge is essential to improve the playing level. A straightforward knowledge-based player such as DB outperforms UCT by 4000 simulations (70%). We have

shown how to insert fundamental concepts, such as biggest cell attack, balanced one-color diagram, one-round game heuristic, into UCT, yielding successive versions of UCT: BUCT, aBUCT, and ABUCT. The latter one is an aggressive and balanced UCT player using 500 simulations that obtains 93% of wins against UCT with the same number of simulations. As in Go [5], inserting domain-dependent knowledge into the simulations improves the playing level. Our study shows that domain dependent knowledge brings about improvements far better than the improvements brought about by RAVE or GP. To sum up, we have: $UCT \leq RAVE \approx GP \ll DB \leq aBUCT \ll ABUCT$.

Future works are numerous. We mention six ideas. First, make the length of a VG variable; it will bring information about the robustness of our approaches. Second, let the ABUCT player use several friendly balanced diagrams per color instead of one, and several aggressive balanced diagrams per color; the options at some nodes of the tree are then the strategies following these diagrams. Third, smooth the transition between the strategy used in the middle game (UCT with knowledge) and the strategy used in the last moves (minmax search). Then an intermediate strategy keeping the best of both strategies remains to be found. Fourth, compare other approaches optimizing a function in a continuous space such as HOO [7], or progressive widening [9,11] with our light GP approach (section 4). Fifth, define a multi-player VG and test the ability of MCTS on multi-player games with an infinite branching factor. Finally, popularizing the VG to give birth to other artificial VG players is an enjoying perspective.

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

References

1. Ahn, H.-K., Cheng, S.-W., Cheong, O., Golin, M., van Oostrum, R.: Competitive facility location: the Voronoi game. Theoretical Computer Science 310(1-3), 457–467 (2004)
2. Anuth, J.: Strategien fur das Voronoi-spiel. Master's thesis, FernUniveristät in Hagen (July 2007)
3. Auer, P., Ortner, R., Szepesvári, C.: Improved Rates for the Stochastic Continuum-Armed Bandit Problem. In: Bshouty, N., Gentile, C. (eds.) COLT 2007. LNCS (LNAI), vol. 4539, pp. 454–468. Springer, Heidelberg (2007)
4. Aurenhammer, F.: Voronoi diagrams: a survey of a fundamental geometric data structure. ACM Computing Surveys 23(3), 345–405 (1991)
5. Bouzy, B.: Associating domain-dependent knowledge and Monte-Carlo approaches within a go playing program. Information Sciences 175(4), 247–257 (2005)
6. Brochu, E., Cora, V., de Freitas, N.: A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. Technical Report 23, Univ. of Brit. Col. (2009)
7. Bubeck, S., Munos, R., Stoltz, G., Szepesvári, C.: X-armed bandits. Journal of Machine Learning Research 12, 1655–1695 (2011)

8. Chaslot, G.: Monte-Carlo Tree Search. PhD thesis, Maastricht Univ (2010)
9. Chaslot, G., Winands, M., van den Herik, J., Uiterwijk, J., Bouzy, B.: Progressive strategies for Monte-Carlo tree search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
10. Cheong, O., Har-Peled, S., Linial, N., Matoušek, J.: The one-round Voronoi game. In: 18th Symposium on Computational Geometry, pp. 97–101. ACM (2002)
11. Couëtoux, A., Hoock, J.-B., Sokolovska, N., Teytaud, O., Bonnard, N.: Continuous Upper Confidence Trees. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 433–445. Springer, Heidelberg (2011)
12. 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)
13. Dehne, F., Klein, R., Seidel, R.: Maximizing a Voronoi Region: The Convex Case. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 624–634. Springer, Heidelberg (2002)
14. Faidley, M., Poultney, C., Shasha, D.: The Voronoi game, http://home.dti.net/crispy/Voronoi.html
15. Fekete, S.P., Meijer, H.: The one-round Voronoi game replayed. Computational Geometry Theory Appl. 30, 81–94 (2005)
16. Fortune, S.: A sweepline algorithm for Voronoi diagrams. Algorithmica 2(2), 153–174 (1987)
17. Gelly, S., Silver, D.: Achieving master level play in 9x9 computer go. In: AAAI, pp. 1537–1540 (2008)
18. Guibas, L.J., Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In: 15th ACM Symposium on Theory Of Computing, pp. 221–234. ACM (1983)
19. Kleinberg, R.: Nearly tight bounds for the continuum-armed bandit problem. In: NIPS 17, pp. 697–704. MIT Press (2005)
20. 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)
21. Lloyd, S.P.: Least squares quantization in PCM. IEEE Transactions on Information Theory 28, 129–137 (1982)
22. Mostafavi, M.A., Gold, C., Dakowicz, M.: Delete and insert operations in Voronoi/Delaunay methods and applications. Computers and Geosciences 29(4), 523–530 (2003)
23. Edward Rasmussen, C., Williams, C.K.I.: GaussianProcesses for Machine Learning. MIT Press (2006)
24. Selimi, I.: The Voronoi game (2008), http://www.voronoigame.com/
25. Shewchuk, J.: Triangle: Engineering a 2d Quality Mesh Generator and Delaunay Triangulator. In: Lin, M.C., Manocha, D. (eds.) FCRC-WS 1996 and WACG 1996. LNCS, vol. 1148, pp. 203–222. Springer, Heidelberg (1996)

4*4-Pattern and Bayesian Learning in Monte-Carlo Go*

Jiao Wang, Shiyuan Li, Jitong Chen, Xin Wei, Huizhan Lv, and Xinhe Xu

College of Information Science and Engineering Northeastern University
wangjiao@ise.neu.edu.cn

Abstract. The paper proposes a new model of pattern, namely the 4*4-Pattern, to improve MCTS (Monte-Carlo Tree Search) in computer Go. A 4*4-Pattern provides a larger coverage space and more essential information than the original 3*3-Pattern. Nevertheless the latter is currently widely used. Due to the lack of a central symmetry, it takes greater challenges to apply a 4*4-Pattern compared to a 3*3-Pattern. Many details of a 4*4-Pattern implementation are presented, including classification, multiple matching, coding sequences, and fast lookup. Additionally, Bayesian 4*4-Pattern learning is introduced, and 4*4-Pattern libraries are automatically generated from a vast amount of professional game records. The results of our experiments show that the use of 4*4-Patterns can improve MCTS in 19*19 Go to some extent, in particular when supported by 4*4-Pattern libraries generated by Bayesian learning.

1 Introduction

Go is an ancient board game for two players; it originated in China over 2000 years ago. The game still enjoys a great popularity all over the world [11]. Go has long been considered as the most difficult challenge in the field of Artificial Intelligence and is considerably more difficult than Chess [2]. Given the abundance of problems, and the diversity of possible solutions, computer Go is an attractive research domain for Artificial Intelligence.

Computer Go began in the 1960s with the prevailing static method preferred during the early days. This method chooses a handful of appropriate moves combined them with fast localized tactical searches, see, e.g., GNU GO [8]. Recently, some advanced theories led to a breakthrough performance in computer Go [6], e.g., by Monte-Carlo Tree Search and the Upper Confidence bound for Trees (UCT). At present, the best Go programs running on a cluster are ranked as 2 dan-3kyu.

Currently, the research on enhancements of the MCTS implementation mainly focuses on three key areas, i.e., tree search, random simulation games, and machine learning [12]. Some heuristic algorithms and pruning algorithms, as well as the domain knowledge enhancement methods are described in [4]. The formulation and the use of a pattern is a well-known technique in computer Go [14]. An example is the program GNU GO with its handcrafted pattern database for move selection. Patterns

* The material in this paper is based upon work supported by the NSFC-MSRA Joint Research Fund under Grant 60971057.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 108–120, 2012.
© Springer-Verlag Berlin Heidelberg 2012

are also used in MCTS programs to improve the quality of the random simulation games [5], e.g., in MOGO and FUEGO. The typical patterns applied in MCTS are handcrafted 3*3-Patterns with many limitations.

A novel 4*4-Pattern model is proposed in this paper. It can be easily implemented in random simulation games and generated by Bayesian learning from professional game records. Experimental results show that the 4*4-Pattern is much better than 3*3-Pattern. We consider it an improvement of MCTS.

The paper is organized as follows. Section 2 analyzes the possibility of 4*4-Patterns and then introduces its basic idea. Section 3 describes the necessary operations of the 4*4-Patterns. Additionally, offline 4*4-Pattern-learning based on the Bayesian method is introduced in Section 4. Section 5 shows some experimental results of the 4*4-Patterns. Finally, the conclusion is presented in Section 6.

2 Motivation

Below we describe the 3*3-Pattern background (2.1) and the possibility of 4*4-Patterns.

2.1 The 3*3-Pattern Background

The 3*3-Pattern is nowadays widely used in the move generator in random games. They can improve the quality of random games to some extent so as to enhance the overall performance of the UCT search. In their implementation, the 3*3-Pattern in MOGO is handcrafted [5], whereas FUEGO adopts some hard-coded disciplines [3]. Some examples of 3*3-Patterns are shown below.

Fig. 1. Two examples of a 3*3-Pattern. The left one is the pattern with the move in center of the board. In the right one, the move is on the board edge.

As can be seen in Fig. 1, a 3*3-Pattern is quite straightforward. However, the coverage space of a 3*3-Pattern is limited. Thus, the information provided is meager when considering the huge board space. For example, some classic situations, such as a jump or a diagonal move are inextricable by a 3*3-Pattern due to the space limitation.

2.2 Possibility of 4*4-Patterns

We discuss two items in particular: (1) memory limitations and (2) multiple matching.

- **Memory Limitations**

The major limitation of a 3*3-Pattern and the central symmetry characteristic would suggest expanding the area to a 5*5-Pattern. However, the coverage space of a

5*5-Pattern contains in total 24 points except the central point. Each point has three possible status, i.e., empty point, black point, and white point, so a 5*5-Pattern requires at least 3^{24} bits memory space (approximately 33G byte) to store all the information. The amount of memory demand is not affordable for common computers, so the external disk memory has to be used comparable to an endgame database in Chess [13]. Nevertheless, it is not a preferable way because the high frequency reading may immensely reduce the efficiency of random games, which are executed thousands of times in UCT search [9]. So, we may conclude that a 5*5-Pattern is not applicable on personal computers at present.

Considering the coverage-space defect of a 3*3-Pattern and the memory limitations of the 5*5-Pattern, this paper proposes a new 4*4-Pattern model as a compromising solution. The storage and operations of a 4*4-Pattern is quite special compared to the 3*3-Pattern and the 5*5-Pattern. Once these crucial problems are solved, a 4*4-Pattern can be a desirable improvement in UCT search, which provides a larger area than a 3*3-Pattern and costs less memory than a 5*5-Pattern. In implementations, a single 4*4-Pattern library takes up approximately 14M byte memory, which is acceptable for most common computers.

- **Multiple Matching**

A 4*4-Pattern is not centrally symmetric, thus the traditional mapping method is not applicable for a 4*4-Pattern. To overcome this obstacle, a new method named multiple matching is proposed using multiple templates.

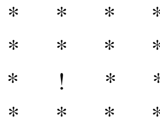


Fig. 2. Match template of 4*4-Pattern

The procedure of multiple matching is explained below. First, traverse all the eight points around the last move (the same as in the 3*3-Pattern procedure), and then apply several different templates on every point for matching. Here, the point is named anchor point, which comes from the Go terminology. There are three categories in total, i.e., center pattern, edge pattern, and corner pattern. All of them have several corresponding fixed templates. Every template reflects to a specific coding order of the 15 stones in the 4*4 area except the anchor point. Third, the coded numeric value is used to query the corresponding pattern library. Fig. 2 shows one of four templates with a center pattern. The "!" means the anchor point, and "*" is the point needed to be coded which may be empty or occupied by a black piece or a white piece.

Compared to a 5*5-Pattern, the coverage space of a single 4*4-Pattern is smaller, exactly 9 points less, and thus carries less information. But the multiple matching with several templates would compensate it to a large extent. For instance, all the points of the 5*5-area around the anchor point are taken into account after four templates have matched with a center pattern. Although a 4*4-Pattern cannot fully achieve the effect

of a 5*5-Pattern, the achievement is still impressive with much lower cost. From another point of view, each pattern is an improvement in the move selection in UCT search even if it is not a perfect one. We remark that even a 5*5-Pattern may suffer from exceptional peripheral information and lead to a wrong decision. But in most cases, the information provided by a pattern is correct and meaningful.

3 Operations of a 4*4 Pattern

This section discusses the required operations of a 4*4-Pattern. In 3.1 we introduce the three categories of a 4*4-Pattern based on the position of the anchor point. In 3.2 we briefly describe the compression to be applied. In 3.3 we discuss the storage and data structure of a 4*4-Pattern library. Specific coding sequences of templates are introduced in 3.4. In 3.5 we provide pseudo codes of crucial operations.

3.1 Classification of the 4*4 Patterns

According to different positions of the anchor point, a 4*4-Pattern can be categorized into three types, i.e., center-pattern, edge-pattern, and corner-pattern. The corner-pattern deals with situations where the anchor point is one of the four corner points on the board. The situation where the anchor point is on the edge points, but not in the corner points, is dealt with by the edge-patterns. The center-pattern deals with all the remaining situations, which are the majority in all situations.

```
*  *  *  *     *  *  *  *     *  *  *  *     *  *  *  *
*  *  *  *     *  *  *  *     *  !  *  *     *  *  !  *
*  !  *  *     *  *  !  *     *  *  *  *     *  *  *  *
*  *  *  *     *  *  *  *     *  *  *  *     *  *  *  *
```

Fig. 3. Center-pattern templates

```
*  *  *  *     *  *  *  *
*  *  *  *     *  *  *  *
*  *  *  *     *  *  *  *
*  !  *  *     *  *  !  *
```

Fig. 4. Edge-pattern templates

```
*  *  *  *
*  *  *  *
*  *  *  *
!  *  *  *
```

Fig. 5. Corner-pattern template

The meaning of the symbols is explained in 2.2. As shown in the Figs 3 to 5, the center-pattern has four templates, whereas the edge-pattern has two and the corner-pattern has only one template.

3.2 Compression

For the edge-pattern and the corner-pattern, the templates are the result after compression. In fact, each of the templates of the edge-pattern includes four situations, i.e., the top edge, the bottom edge, the left-most edge, and the right-most edge. So, there are totally eight templates for the edge-pattern, and it is interesting that some templates are essentially equivalent given some tricks applied on the coding sequence. Fig. 6 shows a compression example of the edge-pattern.

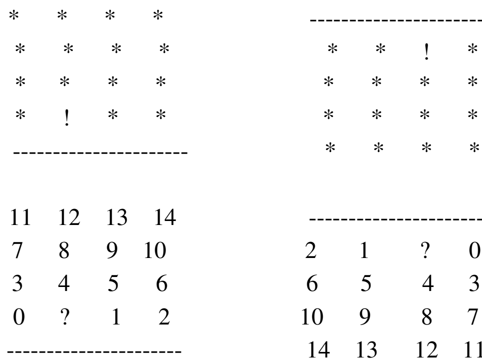


Fig. 6. Example of edge-pattern compression

In the figure, “--“ indicates the boundary of the board, and the two templates on the top are essentially equivalent if the viewing angle turns 180 degree. This can be accomplished by imposing restrictions on the coding sequences. See the following two examples. It is possible to compress all the three 4*4-Pattern types, but in the usual implementations, this is not applied in the center-pattern considering the peripheral disturbance nearby the boundary.

A second method of compression is the color-based method. In a 4*4-Pattern matching, every piece is either white or black; so, all the patterns have two copies, and therefore the current playing side is taken into consideration. The pieces are treated as having the same color as the playing side, or just the contrary. So, we do not use anymore Black or White. Thus, a saving of a halve is achieved by the compression ratio method.

3.3 4*4-Pattern Library

As discussed in 3.1, there are four templates in the center-pattern, two in edge-pattern, and one in the corner-pattern, totally seven. Every template has the same memory occupancy. Three two-dimensional arrays are used for storing the templates, as represented below.

$$\begin{aligned} &\textit{bool CenterTable[4][14348907]} \\ &\textit{bool EdgeTable[2][14348907]} \\ &\textit{bool CornerTable[1][14348907]} \end{aligned} \tag{1}$$

In these arrays, the number 14348907 (3^{15}) represents the maximum possible coding value of 15 points. The first dimension of the array indicates the serial number of the templates, and the second dimension is the coding result of 15 points in the 4*4 area except for the anchor point. It needs to be mentioned that the CornerTable is a linear array in practice, represented as a double-dimension so as to keep the formats in accordancy. The type of data stored in these arrays is “bool”, where true or false indicates whether the corresponding pattern can be chosen or not. The memory occupancy space is nearly 13.7M byte for every pattern library. So, in total 96M byte memory is required for all the pattern libraries. It is affordable for most contemporary computers.

3.4 Coding Sequence and Lookup Table

The query input of a 4*4-Pattern is composed of the piece distribution information of the 5*5-area on the board. For the sake of compression and distinction, strict regulations are made on the coding sequence of the points. The coding sequence rules of all conditions in the three categories are shown in Table 1.

Table 1. Coding Sequence of 4*4 Pattern

Center-Pattern	Serial	0	1	2	3
	Coding Sequence	8 9 10 11 5 6 7 12 3 ? 4 13 0 1 2 14	11 10 9 8 12 5 6 7 13 3 ? 4 14 0 1 2	5 6 7 14 3 ? 4 13 0 1 2 12 8 9 10 11	14 5 6 7 13 3 ? 4 12 0 1 2 11 10 9 8
Edge-Pattern	Serial	0	0	0	0
	Coding Sequence	11 12 13 14 7 8 9 10 3 4 5 6 0 ? 1 2	2 1 ? 0 6 5 4 3 10 9 8 7 14 13 12 11	0 3 7 11 ? 4 8 12 1 5 9 13 2 6 10 14	14 10 6 2 13 9 5 1 12 8 4 ? 11 7 3 0
Edge-Pattern	Serial	1	1	1	1
	Coding Sequence	11 12 13 14 7 8 9 10 3 4 5 6 0 1 ? 2	2 ? 1 0 6 5 4 3 10 9 8 7 14 13 12 11	0 3 7 11 1 4 8 12 ? 5 9 13 2 6 10 14	14 10 6 2 13 9 5 ? 12 8 4 1 11 7 3 0
Corner-Pattern	Serial	0	0	0	0
	Coding Sequence	11 12 13 14 7 8 9 10 3 4 5 6 ? 0 1 2	14 10 6 2 13 9 5 1 12 8 4 0 11 7 3 ?	? 3 7 11 0 4 8 12 1 5 9 13 2 6 10 14	2 1 0 ? 6 5 4 3 10 9 8 7 14 13 12 11

For the sake of saving repeated computation time in multiple matching, the coding sequences of four templates of the center-pattern are rather special. The code of the eight nearest neighbors around the anchor point is calculated only once and the result

is reused subsequently. The coding sequence may seem a little complex, but it is not hard to implement using preset tables.

3.5 Program Codes for Querying

Some pseudo codes of the key procedure in querying are based on the lookup tables given in this subsection. The input is one coordinate of the eight neighbors around the last move of the opponent. The binary output represents whether the point can be played or not.

Some program codes for a 4*4-Pattern querying.

```
bool Match44Any(SgPoint p)
{
  if (IsCenter(p) > 1)
     return MatchAny44Center(p);
  else if (IsEdge (p) > 1)
     return MatchAny44Edge(p);
  else
     return MatchAny44Corner(p);

  return false;
}

//Multiple matching procedure for center-pattern.
bool MatchAny44Center(const BOARD& bd, SgPoint p)
{
  //Caculate the common code of 8 neighbors.
  int cm = CodeOf8CommonNeighbors(m_bd, p);//common code
  /*Iterate 4 templates, return true if the matched pattern is
  favorable, otherwise false. CodeOfRestNeighbors is to
  caculate the codes of the rest 7 neighbors.*/
  if (lookupCenterTable[0][p][0] != INVALID //Table is avaiable
  &&m_44Centertable[0][CodeOfRestNeighbors(m_bd,p,0)+cm]==true)
     return true;
  if (lookupCenterTable[1][p][0] != INVALID
  &&m_44Centertable[1][CodeOfRestNeighbors(m_bd,p,1)+cm]==true)
     return true;
  if (lookupCenterTable[2][p][0] != INVALID
  &&m_44Centertable[2][CodeOfRestNeighbors(m_bd,p,2)+cm]==true)
     return true;
  if (lookupCenterTable[3][p][0] != INVALID
  &&m_44Centertable[3][CodeOfRestNeighbors(m_bd,p,3)+cm]==true)
     return true;

  return false;
}
```

Not all the pseudo codes are shown, such as the functions for edge-pattern and corner-pattern. However, they are quite similar to the ones of the center-pattern, and can be easily implemented.

4 Bayesian Learning of 4*4-Pattern

This section introduces Bayesian learning on 4*4-Pattern, which is a kind of statistical learning. In 4.1 we briefly describe the Bayesian theory and the model designed. In 4.2 we introduce some improvements on the traditional learning process. Then, in 4.3 the learning results are analyzed.

4.1 Bayesian Pattern Learning Model

Bayesian statistics is a classic theory of statistical learning, in which the post probability is calculated from a Bayesian formula combined with the prior probability and conditional probability in discrete condition. The post probability is used for classification instead of the prior probability, because it has more information to reflect the uncertainty of assessing an observation. The Bayesian formula is well known [7]. Bayesian learning has already been adapted in computer Go in recent years. Bruno Bouzy uses Bayesian learning in K-Nearest-Neighbor patterns [1], while David Stern et al. predict the professional moves [10]. An effective offline Bayesian learning model on 4*4-Patterns is proposed according to successful research achievements, by reading every position in professional game records.

$$P_{posterior} = play_time / match_time \tag{2}$$

In the formula, play_time stands for the times a certain pattern is played, while match_time represents that pattern occurrence in time. In a static position, many valid patterns probably exist but only one pattern can be executed. So, the match_time of all the valid patterns increases by one, and play_time of the played patterns increases by one providing that the move matches a specific pattern. For a 4*4-Pattern, every point has to count for several templates when traversing all the points on the board.

4.2 The Improvement of Learning Procedure

The 4*4-Pattern can be automatically generated according to the work (see 4.1), but the learning results are not satisfying. More meaningful improvements should be introduced to make the results better. Below we discuss three suggestions: data preprocessing, adjusting the learning process, and filtering bad patterns.

- **Data Preprocessing**

The quality of the professional game records is vital for learning. Dirty data may originate from the unequal matches, or from a weak game procedure. Some restrictions should be imposed to guarantee the data quality. We mention three of them.

(1) Restriction on the players' level. The level of players can be found by analyzing the SGF files; only those game records are acceptable when the grading of the two players is beyond 6 Dan.

(2) Restriction on the game result gap. The professional game records are accepted only if the result of professional games not exceeds 30 moyo.
(3) Restriction on the winning side. Only the moves of the winning side are input into Bayesian learning program.

By the restrictions above, about 20% of the SGF game records are removed from the game records database and even more samples are eliminated from the sample set. At least, so, the quality of learning material is guaranteed.

- **Adjust the Learning Process**

For a single professional game record, all the moves are input and then executed in proper sequence. Not only should a rule judgment be made to guarantee the correctness, but also attention should be given to some special situation that has to be coped with. The typical one is the move taking pieces, and this situation should avoid pattern learning. Since the pattern is essentially tactical, and not meant for an attacking purpose. Threat or attack is already solved before querying the pattern library during the move generation in a random simulation.

- **Filter Bad Patterns**

There are still many unreasonable patterns available even after the two procedures above. Additional filtering procedures are necessary. Below we mention two of them.

(1) Eliminate the patterns with low post probability. Using post probability as the confidence level is the essence of Bayesian statistics. So, these patterns with low post probability are obviously unacceptable. Currently the minimum of post probability is 5%.
(2) Eliminate the patterns with low match_time or play_time. For example, some arbitrary moves from inspiration are executed once they appeared, so play_time and match_time are all equal to 1 and 100% post probability is obtained. Obviously, it is against the original thoughts of Bayesian statistics. Currently, the limitations of the total amount for both are not less than 10.

5 Experiments

The experiments are composed in two parts, i.e., (1) Bayesian 4*4-Pattern learning experiments and (2) the effectiveness experiments presented below.

5.1 Bayesian 4*4-Pattern Learning Experiments

Two experiments are designed to analyze the result of the Bayesian 4*4-Pattern learning. Over 100,000 professional games are collected for the experiments and the setting of the learning restriction was seen in 4.2. In the first experiment, the game records are input into the learning program one by one, and the statistics of the occupancy rate are kept. For a single pattern library, the occupancy rate is equal to the valid patterns number divide 3^15. The experimental results are shown in Fig. 7.

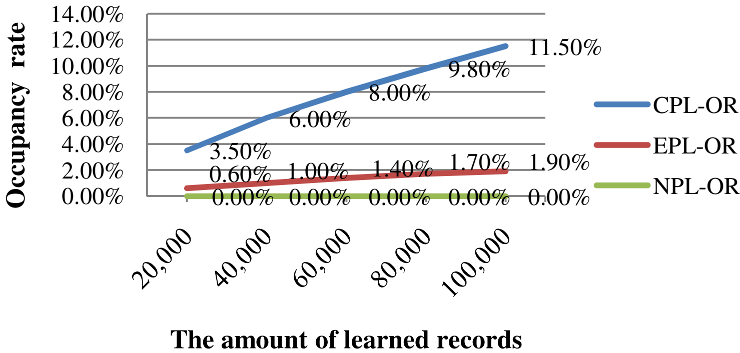


Fig. 7. The occupancy rate of each pattern library

Here CPL-OR means the center-pattern libraries' occupancy rate, while EPL-OR and NPL-OR is the occupancy rate of for edge-pattern libraries and corner-pattern library. It is should be noted that the four libraries of the center-patterns, and also the two libraries in the edge-patterns, share an almost identical distribution of occupancy rate. So, only the typical curves are provided. As can be seen in Figure 7, the occupancy rate goes up while the number of input game records increases, and the CPL-OR reaches 11.50% when all the game records are learned.

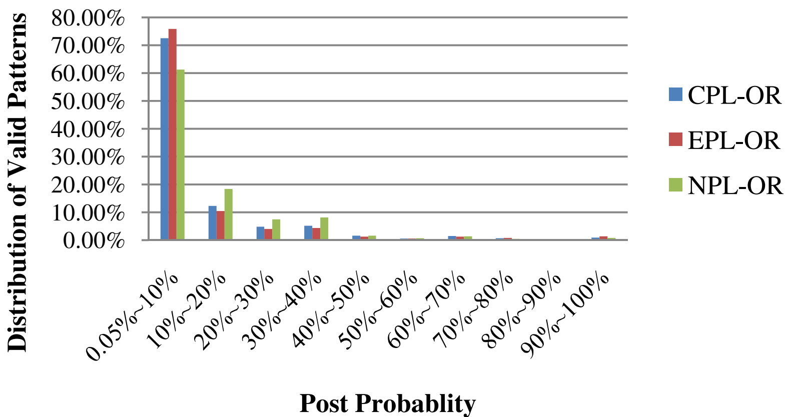


Fig. 8. The distribution of valid patterns according to the value of post probability

The results of the second experiment are shown in Fig. 8. It shows the relationship of post probability and valid patterns. The post probability of the majority of valid patterns is under 10%, and decreases while the percentage range of values rises. Only a few patterns are considered absolutely good, namely that the post probability is 100%. Similar to the former experiment, the sub-libraries also shares an almost identical distribution in the center-pattern and edge-pattern libraries.

5.2 Effectiveness Experiments

In this section, the effectiveness experiments are designed to prove the enhancement of a 4*4-Pattern, and also the correlation with the supporting libraries. The experiments are based on FUEGO (Version 1.1.2), one of the strongest Go programs, which is open-source under GNU license and equipped with a hard-coded 3*3-Pattern in their random simulation. To fit our experiments, the relevant codes of the 3*3-Patterns are removed and the code supporting the 4*4-Patterns is added in FUEGO, together with several sets of all the necessary 4*4-Pattern libraries from the Bayesian 4*4-Pattern learning. Of course, the amount of professional game records is different. To achieve the 4*4-Pattern libraries was easy. They were used in the program by reading external files at their initialization. The 4*4-Pattern program played 1000 games against the original FUEGO, with alternating the playing side. All games used Chinese scoring, 7.5 points Komi and 60 seconds for every move, running on the servers with 4-core Intel i5 2.8Ghz, 4G memory.

The experiments were applied on 19*19 Go and 9*9 Go. Although the 4*4-Pattern libraries were learned from records of 19*19 Go, they still could be used in 9*9 Go. The effectiveness of the experimental results is shown in Fig. 9.

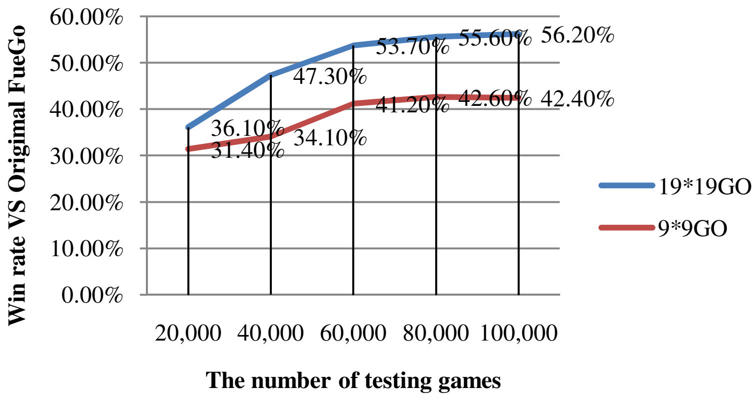


Fig. 9. Effectiveness experiments on 19*19 Go and 9*9 Go with different 4*4-Pattern libraries

As seen in Fig. 9, the playing strength boosts while the amount of learned records increases. However, if the game records for learning are insufficient, the playing strength is unsatisfying due to the low quality of the 4*4-Pattern libraries. The win rate is stable and exceeds 50% once the amount exceeds 60,000. For 9*9 Go, it is amazing that the win rate of the 4*4-Pattern program is always lower than the original FUEGO. There may be two possible reasons. First, the pattern libraries are generated from offline learning by 19*19 Go due to inadequate professional game records of 9*9. Many learned patterns may not be significant in 9*9 Go, because the patterns are more likely to reach the board border. Second, the 19*19 Go is more tactical than 9*9 Go, and a pattern move is mostly a tactical move. So, the effectiveness of the 4*4-Pattern decreases in sharp 9*9 Go games.

A second impact factor of effectiveness is the time limitation. The effectiveness of a 4*4-Pattern is more notable in longer games. The underlying reason is that a

;4*4-Pattern slows down the simulation games to some extent and negatively influences MCTS, but the 4*4-Pattern provides more significant information in a larger coverage space compared to the 3*3-Pattern. Therefore, the contribution of a 4*4-Pattern plays a bigger role than any negative effect. So, the overall effectiveness is positive in the longer games, assuming that the number of simulation games is sufficient.

6 Conclusion and Future Work

In this paper we proposed the 4*4-Pattern model. The details of the implementation are introduced, including design, classification, multiple matching, and coding sequences. In addition, Bayesian learning of 4*4-Patterns and some improvements on the basic method are described. The experimental results show that the 4*4-Pattern is better than the 3*3-Pattern in improving the MCTS in 19*19 Go to some extent, especially in the long games. There are several essential factors for the effectiveness of 4*4-Pattern, i.e., board space, the amount of learned records, time limitation, the effect on different pattern sizes, and the threshold of learning filtration. Some of them are not discussed in this paper, because of the paper length limitation.

Future work should focus on two issues. First, more effective 4*4-Pattern operations require intensive research. In fact, the ideal 4*4-Pattern is not realized unless all the points of 5*5-area around the last opponent move are traversed, and this inevitably costs more time. So, the fast computation and early refutation algorithm are in demand. Second, the learning methods on professional game records should be improved. Bayesian learning is fundamental in statistical learning and the implementation is too straightforward to obtain a convincing gamma value as happened in some top programs. Although the experimental results are satisfying, there is much room for improvement if more appropriate models are adopted.

References

1. Bouzy, B., Chaslot, G.: Bayesian generation and integration of k-nearest-neighbor patterns for 19×19 Go. Computational Intelligence in Games, 176–181 (2005)
2. Bouzy, B., Cazenave, T.: Computer go: An AI oriented survey. Artificial Intelligence 132(1), 39–103 (2001)
3. Fuego Developer's Documentation, `http://www.cs.ualberta.ca/~games/go/fuego/fuegodoc/`
4. Gelly, S., Silver, D.: Combining Offline and Online Knowledge in UCT. In: ICML 2007: Proceedings of the 24th International Conference on Machine Learning, pp. 273–280. Association for Computing Machinery (2007)
5. Gelly, S., et al.: Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062. INRIA,France (2006)
6. Gelly, S., Wang, Y.: Exploration exploitation in go: UCT for Monte-Carlo go. In: On-line trading of Exploration and Exploitation Workshop (2006)
7. Minka, T.P.: A family of algorithms for approximate Bayesian inference. Massachusetts Institute of Technology (2001)
8. Müller, M.: Position Evaluation in Computer Go. ICGA Journal, pp. 219-228 (2002)

9. Silver, D., Tesauro, G.: Monte-Carlo Simulation Balancing. In: Proceedings of the 26th Annual International Conference on Machine Learning, Montreal, Quebec, Canada, pp. 954–852 (2009)
10. Stern, D., Herbrich, R., Graepel, T.: Bayesian Pattern Ranking for Move Prediction in the Game of Go. In: The 23rd International Conference on Machine Learning, pp.873–880 (2006)
11. Stern, D., Graepel, T., MacKay, D.: Modelling Uncertainty in The Game of Go. In: Advances in Neural Information Processing Systems, pp.33–40 (2004)
12. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, pp. 175–182 (2007)
13. Wu, R., Beal, D.F.: A Memory Efficient Retrograde Algorithm and Its Application To Chess Endgames. In: More Games of No Chance, vol. 42. MSRI Publication (2002)
14. Zobrist: Feature. Extraction and Representation for Pattern Recognition and the Game of Go. PhD thesis, University of Wisconsin (1970)

Temporal Difference Learning for Connect6

I-Chen Wu, Hsin-Ti Tsai, Hung-Hsuan Lin, Yi-Shan Lin,
Chieh-Min Chang, and Ping-Hung Lin

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan
{icwu,fs751130,stanleylin,yslin,aup,bhlin}@java.csie.nctu.edu.tw

Abstract. In this paper, we apply temporal difference (TD) learning to Connect6, and successfully use TD(0) to improve the strength of a Connect6 program, NCTU6. The program won several computer Connect6 tournaments and also many man-machine Connect6 tournaments from 2006 to 2011. From our experiments, the best improved version of TD learning achieves about a 58% win rate against the original NCTU6 program. This paper discusses three implementation issues that improve the program. The program has a convincing performance in removing winning/losing moves via threat-space search in TD learning.

1 Introduction

Temporal difference (*TD*) *learning* [13][15], a kind of *reinforcement learning*, is a model-free method for adjusting the state values of the subsequent evaluations. This method has been applied to computer games such as Backgammon [18], Checkers [11], Chess [3], Shogi [4], Go [13] and Chinese Chess [20]. TD learning has been demonstrated to improve world class game-playing programs in the two following cases, TD-GAMMON [18] using TD(λ) and CHINOOK [11] using TDLeaf.

In this paper, we apply TD learning to Connect6, and successfully use TD(0) to improve the strength of a Connect6 program, NCTU6. NCTU6 won the gold medal in the Connect6 tournaments [7][17][23][25] several times from 2006 to 2011, and defeated many top-level human Connect6 players [8][16][29] in man-machine Connect6 championships from 2008 to 2011.

Our experiments showed that the best version of TD learning obtained about a 58% win rate against the original NCTU6 program. The results demonstrated that TD(0) learning can also be used to improve a high-performance world-class game-playing program.

In this paper, we discuss three implementation issues for TD learning, (a) selecting features, (b) removing winning/losing moves (found by threat-space search, which will be described in Section 4), and (c) using the moves played by strong human players for training. Our experiments demonstrate that the issue (b) is quite significant to improve the playing strength of NCTU6.

This paper is organized as follows. Section 2 reviews the game Connect6 and the program NCTU6. Section 3 reviews TD learning including TDLeaf and bootstrapping,

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 121–133, 2012.
© Springer-Verlag Berlin Heidelberg 2012

and describes our design of TD learning for Connect6. Section 4 discusses all the implementation issues of using TD learning, and Section 5 shows the experimental results. Section 6 provides concluding remarks.

2 Connect6 and NCTU6

Connect6 [22][27] is a kind of six-in-a-row game that was introduced by Wu *et al*. Two players, named Black and White in this paper, alternately place two black and white stones respectively on empty intersections of a Go board (a 19×19 board) in each turn. Black plays first and places one stone initially. The first player who obtains six consecutive stones of his own horizontally, vertically, or diagonally wins. The game has been played in one of tournaments held in Computer Olympiads [23][25] (as well as in some other tournaments [7][17]) since 2006.

From [22][27], we know that threats are the key to winning Connect6 (like Go-Moku [1][2] and Renju [10]). According to their definitions, a position is t-threat against the opponent, if and only if t is the smallest number of stones that the opponent needs to place to prevent from losing the game on the next move. A move is called a *1-threat* (also called *single-threat*) *move* if the position after the move is 1-threat, a *2-threat* (*double-threat*) *move* if 2-threat, and a *3-threat* (*triple-threat*) *move* if 3-threat. In Connect6, one player clearly wins by a *3-threat-or-more* move.

In Connect6, many *line patterns* (abbreviated as *patterns* in this paper), such as live-l and dead-l, can grow into threats. As defined in [22][27], live-l (dead-l) of a player can turn into 2-threat (1-threat) if the player places $(4 - l)$ additional stones. For example, live-3 (dead-3) can turn into a 2-threat (1-threat) after *one* additional stone is placed.

In [22][27], a type of winning strategy, called *Victory by Continuous Double-Threat-or-more* moves (*VCDT*) is described. The idea is to win by making continuously double-threat moves and ending by a triple-threat-or-more move or connecting up to six in all variations. It is similar to *Victory by Continuous Four* (*VCF*), a term used in the Renju community [10]. Similarly, the type of winning strategy with additional single-threat moves allowed is called *Victory by Continuous Single-Threat-or-more moves* (*VCST*). In the communities of Connect6 (Renju also), professionals are commonly keen to find these strategies, if there exists any.

Some of the authors developed a lambda-based [19] threat-space search (TSS) technique in [24], named *relevance-zone-oriented proof (RZOP) search*, to find these winning strategies, VCDTs or VCSTs, efficiently and accurately in most of the cases in which there exists any. The RZOP search was incorporated into a Connect6 program, named *NCTU6*, which won several computer Connect6 tournaments and man-machine Connect6 tournaments [7][8][16][17][23][25][29] from 2006 to 2011. When finding no winning strategies, NCTU6 [26] is back to use *alpha-beta search* to find the best move. In the alpha-beta search tree [6], the leaf values are estimated by an evaluation function, and the values of the internal nodes are calculated in the mini-max manner.

In order to make the search more accurate, NCTU6 used the RZOP search [24] to find the winning/losing moves in most nodes in the alpha-beta search. The underlying principle is to avoid choosing losing moves. For extra RZOP search, the averaged time for node evaluation/expansion in alpha-beta search is long. Hence, the number of nodes

in alpha-beta search is relatively small, about 50-500 per second in NCTU6, and the depth of the tree is small too, only about four in NCTU6. Since the alpha-beta search tree is small, a more sophisticated node evaluation/expansion was used to make the search more accurate. In this paper, such a search with heavy node computation is said to be *coarse-grained*. In contrast, strong Chinese-Chess programs (similar to Chess programs) are *fine-grained*, normally expanding about a million nodes per second and searching deeply.

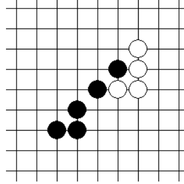


Fig. 1. An example of evaluating *V(s)* for Black

In NCTU6, the evaluation function of positions can be viewed as a function of features, such as threats, live-*l*s and dead-*l*s. Although the function was actually quite complicated, we modified it into a linear combination of features [5] for TD learning. In the example in Figure 1, Black has 1 single-threat, 2 live-2s and 7 live-1s, and White has 1 live-3, 2 dead-2s and 5 live-1s (note that dead-1 is not discussed in this paper for clarity). Given feature weights, NCTU6 evaluates the value of the position for Black in Figure 1 as

$$1 \times w_{T1} + 2 \times w_{L2} + 7 \times w_{L1} + 1 \times w_{-L3} + 2 \times w_{-D2} + 5 \times w_{-L1}$$

where w_f is the weight of feature *f*, which indicates *n*-threat by Tn, live-*n* (dead-*n*) by Ln (Dn), and the opponent's features by $-f$. Note that for 2-threat we use $1 \times w_{T2}$, instead of $2 \times w_{T1}$. Let $\varphi(s)$ denote a vector of feature numbers in a position (or state) s, and θ denote a vector of feature weights. Thus, the value of a position s is

$$V(s) = \varphi(s) \cdot \theta \tag{1}$$

In the example in Figure 1, $\varphi(s) = [1,2,7,1,2,5]$, if the vector of features is $[T1, L2, L1, -L3, -D2, -L1]$.

In the original NCTU6, the weights θ were hand-tuned from some experiences of games against the top-level human players. However, as the number of features grew, it became hard to produce these weights accurately. The goal of this paper is to use TD learning to help adjust these weights automatically.

3 TD Learning for Connect6

This section first reviews TD learning and TDLeaf/bootstrapping in Subsections 3.1 and 3.2 respectively, and then describes our design for TD learning in Subsection 3.3.

3.1 TD Learning

As described in Section 1, TD learning is a kind of reinforcement learning. In TD(0) (see [13][15]), the value function V of a state is used to approximate the expected return, instead of waiting until the complete return has been observed. The error between states s_t and s_{t+1} is $\delta_t = r_{t+1} + V(s_{t+1}) - V(s_t)$, where r_{t+1} is the reward at time $t+1$. In Connect6 as well as some other computer games, the reward, say 1 for winning and –1 for losing, is obtained at the end game, and the reward is zero during the game playing. For clarity, for the end state (or the end game) s_T, let the value of $V(s_T)$ be r_T. Then, the error is simplified as $\delta_t = V(s_{t+1}) - V(s_t)$. The value of $V(s_t)$ in TD(0) is expected to be adjusted by the following value difference $\Delta V(s_t)$,

$$\Delta V(s_t) = \alpha\delta_t = \alpha\big(V(s_{t+1}) - V(s_t)\big) \tag{2}$$

where α is a step-size parameter to control the learning rate. For general TD(λ) (also see [13][15]), the value difference is

$$\Delta V(s_t) = \alpha\left((1-\lambda)\sum_{n=1}^{T-t-1}\lambda^{n-1}V(s_{t+n}) + \lambda^{T-t-1}V(s_T) - V(s_t)\right). \tag{3}$$

Note that the TD(1) learning is similar to the learning with Monte-Carlo tree search.

In order to correct the value $V(s_t)$ by the difference $\Delta V(s_t)$, we can adjust the feature weights θ by a difference $\Delta\theta$ based on $\nabla_\theta V(s_t)$. For linear TD(0) learning, where $V(s_t)$ is linear like formula (1), the difference $\Delta\theta$ is

$$\Delta\theta = \Delta V(s_t)\varphi(s_t) = \alpha\delta_t\varphi(s_t) \tag{4}$$

In order to control the learning rate better, the above difference is modified with normalization, like the NLMS [14], as follows.

$$\Delta\theta = \Delta V(s_t)\frac{\varphi(s_t)}{\|\varphi(s_t)\|^2} = \alpha\delta_t\frac{\varphi(s_t)}{\|\varphi(s_t)\|^2} \tag{5}$$

3.2 TDLeaf and Bootstrapping

The researchers in [3] proposed the so-called TDLeaf to improve the weights of the features for their Chess program KNIGHTCAP. The method is to run the normal alpha-beta search and choose the leaves of the principal variation (PV) for TD learning, instead of the roots. However, as pointed out by [21], the method has the following three drawbacks. First, only one update is used for each search and other information is wasted. Second, the updates are only based on the positions of best play, which may not represent all the moves. Third, the target search is accurate only when both the player and opponent are strong.

In order to solve these problems, the researchers in [21] proposed a new method for bootstrapping from the minimax game-tree search. In their method, for all subtrees of the search, if their PVs are available, nodes on PVs are used for training. Note that their

method can be extended to alpha-beta search. Thus, they update many nodes with PVs in the search tree, rather than a single node, and the outcome of a deep search is used for training, instead of the outcome of a subsequent search.

3.3 Our TD Learning

Although the bootstrapping method seems promising, our design of TD learning for NCTU6 brings us back to the original TD(0) learning as explained below (see the last paragraph). Now, we want to describe the necessity of using a two-ply update, $\Delta V(s_t) = \alpha\big(V(s_{t+2}) - V(s_t)\big)$, instead of a one-ply update as in formula (2). In a one-ply update, updating the nodes between both players may cause overweighting the updates, since the player to move has always one move less (less advantage) than the other. Thus, the phenomenon of overweighting may cause a large fluctuation of the evaluated values. This problem is even more serious for Connect6 (recall two stones per move). Our algorithm for TD(0) is designed based on Silver's (cf. Algorithm 3 in [13]) as follows.

Algorithm TD(0) Learning Applied to NCTU6

```
Procedure TD_Learning(n)
 1: i = 0
 2: while i < n do
 3:    board.Initialize()
 4:    SelfPlay(board)
 5:    i++
 6: end while
end procedure

Procedure SelfPlay(board)
 1: t = 0
 2: V0,φ0 = Eval(board)
 3: while not board.Terminal() do
 4:    at = Greedy(board,ε)
 5:    board.Play(at)
 6:    t++
 7:    Vt, φt = Eval(board)
 8:    if t ≥ 2 then
 9:       δ= Vt - Vt-2
10:       Norm = ||φt-2[i]||^2
11:       for all i ∈φt-2 do
12:          θ[i] += αδφt-2[i] / Norm
13:       end for
14:    end if
15: end while
end procedure

Procedure Greedy(board, ε)
 1: if Bernoulli(ε) = 1 then return Random(board)
 2: if board.BlackToPlay() then
 3:    a* = Pass; V* = 0
 4:    for all a ∈ board.Legal() do
 5:       board.Play(a)
 6:       V = Eval(board)
 7:       if V ≥ V* then
 8:          V* = V; a* = a
 9:       end if
10:       board.Undo()
11:    end for
12: else // omitted for White to play
13: end if
14: return a*
end procedure

Procedure Eval(board)
 1: φ = board.GetFeatures()
 2: v = 0
 3: for all i ∈φ do
 4:    v += φ[i]θ[i]
 5: end for
 6: V = 1/(1+e^-v)
 7: return V, φ
end procedure
```

Our TD learning performs n training games by calling *TD_Learning(n)*. In each training game, we initialize the state (or board) by *board.Initialize*(), which selects initial boards from our database, mainly selected from Little Golem [9]. Then, we call the procedure *SelfPlay(board)*, to make the subsequent moves of a game.

This procedure *SelfPlay(board)* plays on its own by repeatedly calling *Greedy(board,ε)* to make moves. The procedure *Greedy(board,ε)* selects a move

according to the so-called *ε-Greedy policy* [13][15]. The ε-Greedy policy is to play at random with probability ε, as in Line 1 of this procedure, and to play the best move with probability (1-ε), as in Lines 2 to 14 (the case of White to play is omitted). The best move (or action) is chosen among all moves of which the values are determined by the evaluation procedure *Eval*.

The procedure *Eval* returns the state value and the vector of feature numbers of the board. As formula (1), the value function $V(s) = \varphi(s) \cdot \theta$ is evaluated in Lines 1 to 5 and adjusted into the range [0,1] by a function $1/(1+e^{-V(s)})$ in Line 6. In case that Black (White) wins, the procedure *Eval* returns the state value one (zero).

Now, let us compare TD(0) with the bootstrapping method [21]. As described in Section 2, the search tree in NCTU6 is coarse-grained and shallow (the averaged depth of the tree is only about four). Consider the PV in a tree search, $\{s_0, s_1, s_2, s_3, s_4\}$, where s_0 is the root and s_4 is a leaf. Due to two-ply updates, we can only update both from s_4 to s_0 and from s_4 to s_2 in the bootstrapping method. For many of other subtrees, if their PVs are available, say $\{s_1, s_2, s_3, s_4\}$, we can only update from s_4 to s_2. According to our analysis on NCTU6, only about 400 updates can be used for training in an alpha-beta search tree with 10,000 nodes expanded. In contrast, 10,000 updates can be used for training in TD(0), when 10,000 nodes expanded. Thus, TD(0) apparently has more updates than bootstrapping.

4 Implementation Issues

This section discusses three issues when implementing the linear TD(0) learning for Connect6. These issues include (a) selecting features, (b) removing winning/losing moves found by threat-space search, and (c) using moves played by strong human players. These issues are discussed in the following three subsections respectively.

4.1 Feature Selection

As described above, this paper modifies the evaluation function into a linear combination of features, including the types of patterns, the distance of the patterns from the board center (or border), the direction of the pattern, and the game stages.

As described in Section 2, the types of patterns mainly include 1-threat, 2-threat, live-3 to live-1, dead-3 to dead-1, etc. In fact, there are more complex patterns, such as the pattern with live-1 and dead-2 at the same time. For example, the diagonal line containing two white stones in Figure 1 includes both dead-2 and live-1 at the same time. This pattern is actually stronger than dead-2 and live-1. However, for clarity of discussion, such patterns are disregarded in this paper.

Some other important features related to patterns are discussed as follows. The patterns on the border of the board tend to threaten the opponent less. The diagonal line patterns tend to be stronger, since diagonal line patterns normally cover a larger territory for attacking.

Next, we want to investigate features in different stages. Like some other games, such as Chinese Chess, the playing strategies in the three stages, opening, middle-game and end-game, are somewhat different. So, in our TD learning, the moves and features are also treated differently in the three stages. For instance, according to top-level

human players, live-2 in the opening is more important than in the end-game, since threats are not many in the opening.

4.2 Threat Space Search

It is a really important issue to remove winning/losing moves found by threat-space search (TSS). The key reason is that these losing moves are removed and not evaluated in alpha-beta search (like what NCTU6 does). Training these moves becomes noise in learning. In Connect6, it is a well-known strategy for players not to abuse playing 2-threats or 1-threats, if no winning strategies are found yet. Just like Go, beginners are taught not to abuse playing atari. Although the authors of NCTU6 [26] knew the idea upon designing, it was hard to tune the weights by hand. This is also one of the motivations of this paper.

Fig. 2. Avoiding winning/losing paths

If we do not remove these moves, the TD learning tends to make the weights of 1-threats or 2-threats excessively high. In the case that one player wins by a VCDT in the TD learning, the player plays many 2-threat moves in the last moves of the training games as shown in Figure 2. Consequently, 2-threat is wrongly regarded as a rather important feature and therefore adjusted to be overweighed.

In order to solve this problem, we propose to use TSS, the RZOP search [24] also used in NCTU6, to remove those winning/losing moves near the end as above. More specifically, TSS is performed to check winning of the position before running *Greedy* in Line 4 of *SelfPlay*, and once a winning move is found, *SelfPlay* terminates the game and restarts another training game. One minor drawback of this approach is the higher computation time for training, since the times spent on TSS are longer than node evaluation/expansion.

4.3 Learning from the Games Played by Strong Human Players

In Subsection 3.3, our TD learning program uses the procedure *Greedy* to make a move by the ε-Greedy policy. It is also an interesting issue to use the games played by strong human players, instead of using *Greedy*, as in [12]. Namely, in Line 4 of SelfPlay, *Greedy* is replaced by a routine which retrieves moves from the game record. This paper collected the games, about 30,197 games, where at least one of the players was ranked with points higher than 1800 from Little Golem [9]. The collection of records of these games is called *the expert collection* in this paper.

One advantage of using these games for learning is to spend less time on making a move. In order to remove winning/losing moves by TSS, we used a preprocessor to run TSS backwards from the leaf. Normally, TSS runs faster on winning/losing positions than the positions without winning/losing. Since running TSS backwards is performed on at most one position without winning/losing, the computation time is lower than that for running TSS forwards as in the algorithm in Subsection 3.3. A second advantage is to let the program learn to play like these players, if possible. For example, these players usually do not abuse playing 1-threats or 2-threats.

5 Experiments

In this section, we first describe our experimental environment in Subsection 5.1. Then, we analyze our experiments in different aspects, including stages, threat-space search, and training games, which are discussed in Subsections 5.2 to 5.4, respectively. Finally, we summarize and discuss the experimental results in Subsection 5.5.

5.1 Experimental Environment

In our experiments, we used TD(0) learning as shown in the algorithm in Subsection 3.3. We set $\alpha = 0.1$ and $\varepsilon = 0.1$. The number of features in total was about 500.

In our experiments, we measured the strength of the program learned from TD learning like [11] as follows. After finishing TD learning, we replaced the original feature weights of NCTU6 by the trained feature weights. Let *NCTU6-TD* denote the NCTU6 with the newly trained feature weights. In order to compare the strength of NCTU6-TD with that of the original NCTU6, we selected 176 popular openings from the expert collection. Namely, the openings that were played in at least 30 games in the collection. For each selected opening, let NCTU6-TD play twice against NCTU6, one for Black and the other for White, respectively. Thus, NCTU6-TD played 352 games against NCTU6 in total. NCTU6-TD obtained 2 points for a win, 1 for a draw, and nothing for a loss. The win rate was the total obtained points divided by 704, after finishing all the 352 games.

For each experiment of TD learning, all feature weights were initialized to 0 (zero knowledge), and the numbers of training games we ran were 0, 100, 300, 1000, 3000, 10000 and 30000. Since it took long times to do the experiments, we used the volunteer computing system in a job-level manner as described in [28].

5.2 Stages

As mentioned in Subsection 4.1, the playing strategies in the three stages, opening, middle-game, and end-game, are somewhat different. In our experiments, the first 10 moves in a game were considered to be played in the opening, the next 20 moves were in the middle-game, and the rest were in the end-game.

In this subsection, we tried four versions of stages for comparisons. The first version, called *1-stage*, was to have one stage only. The second version, called *3-stage*, was to have three stages as above. Thus, each feature had the different weights in

different stages in 3-stage, but the same in 1-stage. The third version, called *hybrid-3-stage*, was to use the feature weights of the original NCTU6 in opening, but use the feature weights trained by the second version in both middle-game and end-game. The fourth version, called *hybrid-2-stage*, was the same as hybrid-3-stage except that middle-game and end-game are combined into one stage.

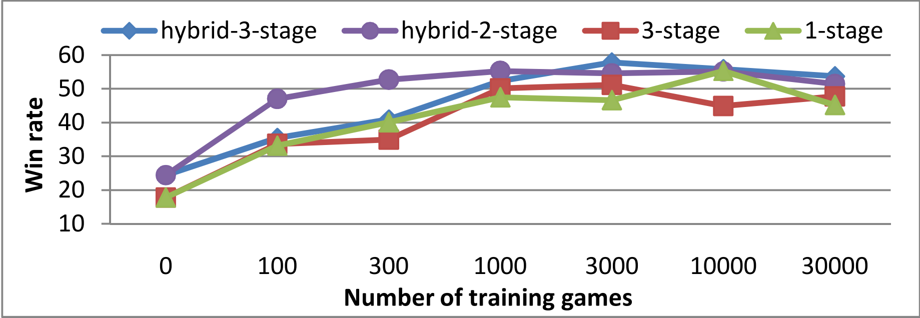


Fig. 3. The win rates for TD learning with different versions of stages

Figure 3 demonstrates the experimental results for the four different versions. In all of these experiments, we removed moves via TSS (namely the RZOP search) as described in Subsection 4.2. The results showed that both hybrid-2-stage and hybrid-3-stage were consistently better than 1-stage and 3-stage. The best was the one after 3000 training games in hybrid-3-stage. In our analysis, we observed two reasons for the phenomenon (of being the best) as follows. First, in opening, the features for those threats or patterns far away from the center were rarely used and therefore trained only few times. Thus, it became hard to learn these feature weights well in the first two versions. Second, for TD(0) learning, it was slow to learn the feature weights in opening since the learning propagation from end-game to opening was slow. Thus, the features in opening were relatively hard to learn.

For both hybrid-2-stage and hybrid-3-stage, hybrid-3-stage performed better for 3000 training games or more, but worse for less than 3000, for the following reason. Since hybrid-3-stage had more features, the learning rate was much slower. However, hybrid-3-stage performed better if there were sufficient training games.

5.3 Threat Space Search

As explained in Subsection 4.2, threat-space search (TSS) is a quite important issue. Figure 4 shows the results for TD learning with and without removing winning/losing moves found by TSS (namely the RZOP search [24]). In all of these experiments, we used hybrid-3-stage as above.

The results demonstrated significant and consistent improvements in all cases. The win rates with TSS (used to remove winning/losing moves) were about at least 7.1% higher than those without TSS. From the results, we may conclude that TSS plays a quite significant role in the TD learning.

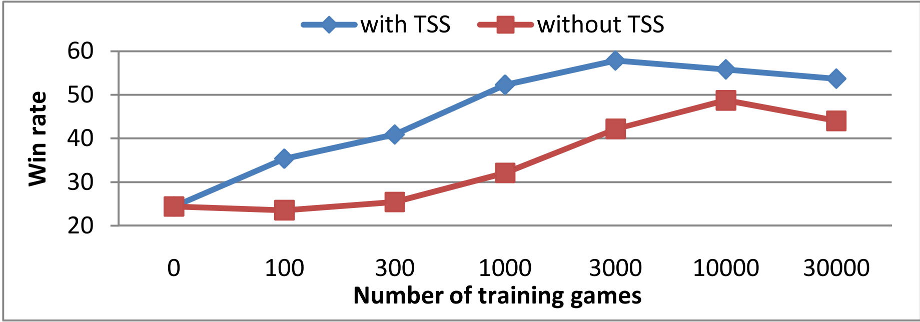


Fig. 4. The win rates for TD learning with and without using TSS

Table 1. The weights of features learned from TD learning with/without TSS

Feature Weights	With TSS	Without TSS
W_{T2}	0.52982	1.73220
W_{T1}	0.51070	0.83796
W_{L3}	0.49358	0.73046
W_{D3}	0.27506	0.25531
W_{L2}	0.20028	0.07715

Table 1 shows the weights of the features, 2-threats, 1-threat, live-3, dead-3, and live-2, learned from the TD learning with and without using TSS. The result clearly shows that the weight of 2-threat was high relatively to others (W_{T2} is nearly double of W_{T1}) in TD learning without TSS.

5.4 Training Games

As described in Subsection 4.3, selecting training games is a delicate issue for TD learning. The experiments in this subsection were done to investigate this issue by considering TD learning that (1) used the game records in the expert collection and that (2) used ε-*Greedy* to generate moves. In addition, for each case, we also considered TD learning with and without TSS (removing winning/losing moves).

Figure 5 showed the results for four kinds of TD learning. In all of these experiments we also used hybrid-3-stage as above. Still, it also showed that the TD learning with TSS was consistently and clearly better than the learning without TSS.

Below we consider using TSS. The results showed that the TD learning with ε-*Greedy* was slightly better than that with the expert collection for 1000 to 10,000 training games. In the case of using 30,000 training games, the TD learning with the expert collection was slightly better. Since the collection included about 30,000 games only, it was unsure about whether the win rate would be higher for more games.

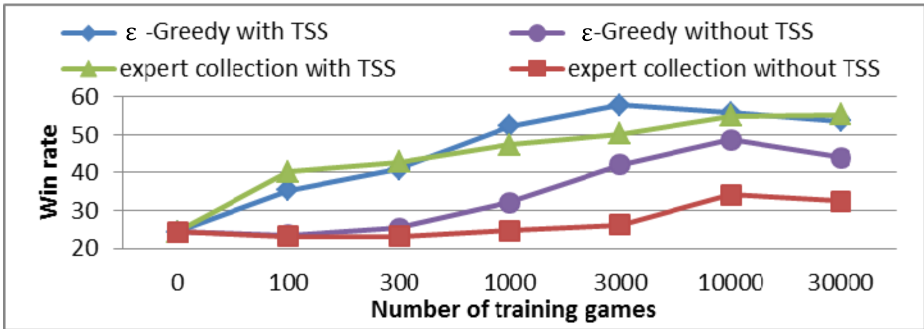


Fig. 5. The win rates for TD learning using ε-Greedy and the expert collection

5.5 Discussion

From Subsections 5.2 to 5.4, we may conclude that TD learning with TSS (used to remove winning/losing moves) is the most important factor to improve the strength. The win rates with TSS were at least 7.1% higher than those without it. Using the version of hybrid-3-stage also helps improve TD learning. The merit of using the expert collection is unclear yet.

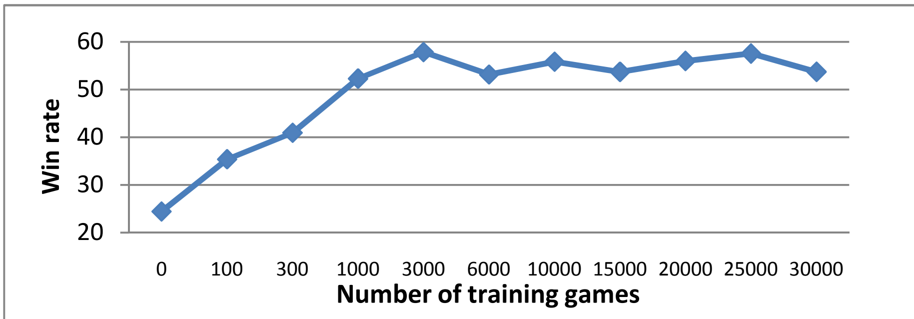


Fig. 6. The win rates for TD learning with more different training games

In the rest of this subsection, we discuss the convergence and computation times for training. In order to see whether TD learning in our experiments converges, we also ran 6000, 15000, 20000 and 25000 training games for the experiment with hybrid-3-stage,ε-Greedy, and TSS. Figure 6 shows that these values converged around 53-58% after running more than 3000 training games.

As for the training times, Table 2 showed the total times spent on training 10,000 games by using TSS or not, and by using the expert collection or not. Apparently, the versions with TSS ran much more slowly than the ones without TSS, due to the extra TSS overhead. Here, we consider the two versions with TSS. The one with the expert collection ran much faster than that with *ε-Greedy*, since we removed winning/losing moves backwards from the leaves in the former, as explained in Subsection 4.2.

Table 2. The comparison of the time spent with/without TSS

TD Learning	Times for 10,000 training games
ε-Greedy with TSS	677 min. (or 11 hr. 17 min.)
ε-Greedy without TSS	31 min.
Expert collection with TSS	32 min.
Expert collection without TSS	2 min.

6 Conclusion

In this paper, we demonstrate a solid application of TD(0) learning for Connect6. We successfully use TD(0) learning to improve the strength of NCTU6, a Connect6 program. Our experiments showed that the best version, improved via our TD learning method, obtained about a 58% win rate against the original NCTU6 program.

This paper also discusses several issues of implementing TD learning. From them we may conclude that TD learning plays a quite important role to remove winning/losing moves found by TSS (namely the RZOP search used in NCTU6). Our experiments demonstrated significant and consistent improvements in all cases. Using the version of hybrid-3-stage also helps improve TD learning. The merit of using the professional collection is unclear.

Although the bootstrapping method was not tried, this paper demonstrated that TD(0) learning worked sufficiently well for NCTU6. From this paper, it is conjectured that TD(0) should also work for other programs with coarse-grained and shallow search trees, though the comparison between TD(0) and bootstrapping is still to be performed in the future.

Acknowledgement. The authors would like to thank the anonymous referees for their valuable comments, and thank the National Science Council of the Republic of China (Taiwan) for financial support of this research under contract numbers NSC 97-2221-E-009-126-MY3, NSC 99-2221-E-009-102-MY3 and NSC 99-2221-E-009-104 -MY3.

References

[1] 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)

[2] Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence, Ph.D. Thesis, University of Limburg, Maastricht (1994)

[3] Baxter, J., Tridgell, A., Weaver, L.: Learning to Play Chess Using Temporal Differences. Machine Learning 40(3), 243–263 (2000)

[4] Beal, D.F., Smith, M.C.: First Results from Using Temporal Difference Learning in Shogi. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 113–125. Springer, Heidelberg (1999)

[5] 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)
[6] Knuth, D.E., Moore, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6, 293–326 (1975)
[7] Lin, H.-H., Sun, D.-J., Wu, I.-C., Yen, S.-J.: The 2010 TAAI Computer-Game Tournaments. ICGA Journal 34(1), 51–55 (2011)
[8] Lin, P.-H., Wu, I.-C.: NCTU6 Wins in the Man-Machine Connect6 Championship 2009. ICGA Journal 32(4), 230–233 (2009)
[9] Golem, L.: Online Connect6 games (2006), `http://www.littlegolem.net/`
[10] Renju International Federation, The International Rules of Renju (1998), `http://www.renju.nu/rifrules.htm`
[11] Schaeffer, J., Hlynka, M., Jussila, V.: Temporal Difference Learning Applied to a High-Performance Game-Playing Program. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence, pp. 529–534 (August 2001)
[12] Schraudolph, N.N., Dayan, P., Sejnowski, T.J.: Learning to Evaluate Go Positions via Temporal Difference Methods. In: Baba, N., Jain, L. (eds.) Computational Intelligence in Games, vol. 62. Springer, Berlin (2001)
[13] Silver, D.: Reinforcement Learning and Simulation-Based Search in Computer Go, Ph.D. Dissertation, Dept. Comput. Sci., Univ. Alberta, Edmonton, AB, Canada (2009)
[14] Simon, H.: Adaptive Filter Theory. Prentice Hall (2002)
[15] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
[16] Taiwan Connect6 Association, Connect6 homepage (2007), `http://www.connect6.org/`
[17] TCGA Association, TCGA Computer Game Tournaments, `http://tcga.ndhu.edu.tw/TCGA2011/`
[18] Tesauro, G.: TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. Neural Computation 6, 215–219 (1994)
[19] Thomsen, T.: Lambda-Search in Game Trees - with Application to Go. ICGA Journal 23, 203–217 (2000)
[20] Trinh, T., Bashi, A., Deshpande, N.: Temporal Difference Learning in Chinese Chess. In: Mira, J., Moonis, A., de Pobil, A.P. (eds.) IEA/AIE 1998. LNCS, vol. 1416, pp. 612–618. Springer, Heidelberg (1998)
[21] Veness, J., Silver, D., Uther, W., Blair, A.: Bootstrapping from Game Tree Search. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., Culotta, A. (eds.), Advances in Neural Information Processing Systems 22. pp. 1937–1945 (2009)
[22] Wu, I.-C., Huang, D.-Y.: A New Family of *k*-in-a-Row 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. 180–194. Springer, Heidelberg (2006)
[23] Wu, I.-C., Lin, P.-H.: NCTU6-Lite Wins Connect6 Tournament. ICGA Journal (SCI) 31(4), 240–243 (2008)
[24] Wu, I.-C., Lin, P.-H.: Relevance-Zone-Oriented Proof Search for Connect6. IEEE Transaction Computer Intelligence AI Games 2(3) (September 2010)
[25] Wu, I.-C., Yen, S.-J.: NCTU6 Wins Connect6 Tournament. ICGA Journal (SCI) 29(3), 157–159 (2006)
[26] Wu, I.-C., et al.: The Search Techniques in NCTU6 (in preparation)
[27] Wu, I.-C., Huang, D.-Y., Chang, H.-C.: Connect6. ICGA Journal 28(4), 234–242 (2006)
[28] Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C., Chen, B.-T.: Job-Level Proof-Number Search for Connect6. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 11–22. Springer, Heidelberg (2011)
[29] Wu, I.-C., Lin, Y.-S., Tsai, H.-T., Lin, P.-H.: The Man-Machine Connect6 Championship 2011. ICGA Journal 34(2), 103–106 (2011)

Improving Temporal Difference Learning Performance in Backgammon Variants

Nikolaos Papahristou and Ioannis Refanidis

University of Macedonia, Department of Applied Informatics,
Egnatia 156, Thessaloniki, 54006, Greece
{nikpapa,yrefanid}@uom.gr

Abstract. *Palamedes* is an ongoing project for building expert playing bots that can play backgammon variants. As in all successful modern backgammon programs, it is based on neural networks trained using temporal difference learning. This paper improves upon the training method that we used in our previous approach for the two backgammon variants popular in Greece and neighboring countries, Plakoto and Fevga. We show that the proposed methods result both in faster learning as well as better performance. We also present insights into the selection of the features in our experiments that can be useful to temporal difference learning in other games as well.

1 Introduction

Backgammon is an ancient board game of luck and skill that is quite popular throughout the world with numerous tournaments and many popular variants. Variants of any game usually are not as interesting as the standard version, but often offer a break in the monotony of playing the same game over and over again. Backgammon variants come in different flavors: some change the standard backgammon rules only slightly (Portes), while others have different rules for moving the checkers (Fevga, Plakoto), alternate starting positions (Nackgammon), have a different checker direction (Fevga) or assign a special value to certain dice rolls (Acey-Deucey, Gul-bara). *Palamedes* [6] (Fig. 1) is an ongoing project dedicated to offer expert-level playing programs for backgammon variants.

The objective for each player of virtually all variants is to move all his[1] checkers to the last quadrant (called the *home board*), so he can start removing them; a process called *bearing off*. The player that removes all his checkers first is the winner of the game. Players may also win a double game (2 points) when no checker of the opponent has been beared-off. A triple win and the doubling cube are normally used only in standard backgammon.

In previous work [7], following the successful example of TD-Gammon [14,15,16] and other top-playing backgammon programs, we trained neural networks (NN) using temporal difference learning for playing *Plakoto* and *Fevga,* two variants

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

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 134–145, 2012.
© Springer-Verlag Berlin Heidelberg 2012

very popular in Greece and the neighboring countries. The contribution of this paper is (1) the improvement of our training methods and (2) the presentation of new results for these games. More specifically, we improved the performance of the training method by making the target of the updates to be the inverted value of the next player's position; we also improved a little the learning speed by updating the positions starting from the end and recalculating the target value. Furthermore, we identified problems with the generalization of the neural network at certain positions of the Plakoto variant. We present our solution based on adjusting the input features along with an analysis of the cause of the problem and its implications to learning from manually added features in general.

The remaining of the section describes the main rules of the Plakoto and Fevga variants and compares the complexity of the games with standard backgammon. The complete set of rules for standard backgammon, Plakoto, Fevga, and other variants can be found in [1].

1.1 Plakoto

The key feature of game Plakoto is the ability to pin hostile checkers, so as to prevent their movement. The general rules of the game are the same as the regular backgammon apart from the procedure of hitting. Players start the game with fifteen checkers placed in opposing corners and move around the board in opposite directions till they reach their home boards which are located opposite of the starting area.

When a checker of a player is alone in a point, the opponent can move a checker of his own in this point thus pinning (or trapping) the opponent's checker. This point counts then as a *made point* as in regular backgammon, which means that the pinning

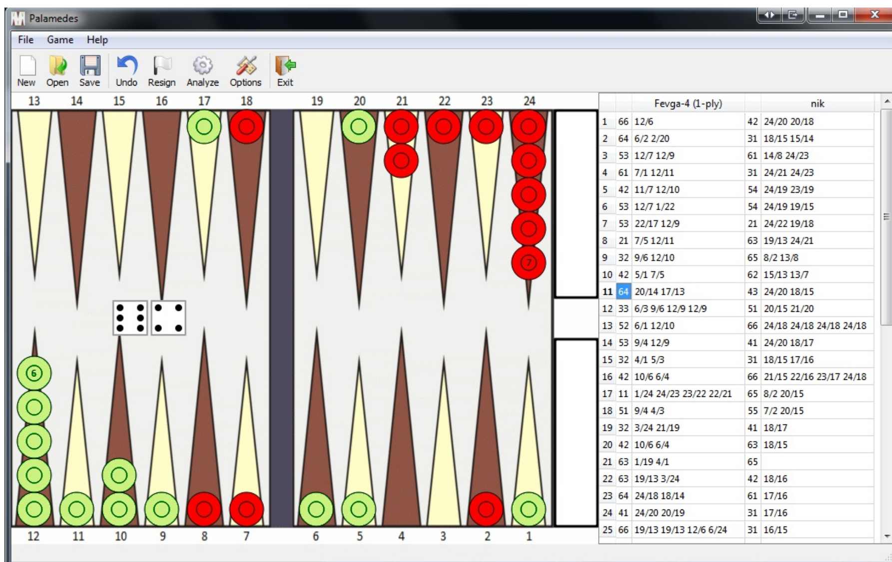


Fig. 1. Palamedes: A program for playing backgammon and variants

player can move checkers in this point while the pinned player cannot. The pinned checker is allowed to move normally only when all opponent pinning checkers have left the point (*unpinning*). Pinning the starting point of the opponent results immediately in a double win.

1.2 Fevga

The main difference of Fevga is that there is no pinning or hitting. If the player has even a single checker in one point, this point counts as a *made point*, effectively preventing the movement of the opponent's checkers in this point. Each player starts with fifteen checkers on the rightmost point of the far side of the board, at diagonally opposite corners from each other, whereas the two players move in the same direction. A crucial characteristic of this variant is that a prime formation (six consecutive made points) can be more easily achieved than in backgammon. Consequently, human experts usually plan their strategies always having in mind (1) to create some kind of prime in order to block the opponent as soon as possible in his development, while at the same time (2) preventing the opponent from blocking them.

1.3 Complexity Compared to Standard Backgammon

Table 1 summarizes the differences in complexity of backgammon, Plakoto, and Fevga. The standard backgammon state space has been estimated as exceeding 10^{20} states [16]. The Fevga variant without having a bar position (a position where the hitted checkers are placed in standard backgammon) inevitably has somewhat fewer states, whereas the possibility of pinning gives Plakoto a higher number of total states. The numbers shown for the average branching factor and the average game length in table 1 were computed by taking the best agent at our disposal for each game and making it play 50,000 games against itself. We used Plakoto-3 for Plakoto, Fevga-5 for Fevga and a NN trained with expert features for backgammon. This backgammon NN has a performance of 0.608 against the pubeval [8] benchmark program.

Table 1. State space size and game tree complexity

Game	State Space Size	Branching Factor (avg)	Game Length (avg)
Backgammon (BG)	$> 10^{20}$	16	55
Plakoto	> BG	23	92
Fevga	< BG	25	91

One common difference of both games compared to standard backgammon is that they last longer: whereas a standard backgammon game lasts on average 55 plies, a game of Fevga and Plakoto lasts on average around 92 plies. The game of Fevga is the most straightforward of the three. Players run their checkers to their home board resulting in game sequences with more or less the same total ply count. In standard backgammon the possibility of hitting sometimes results in long sequences. In Plakoto the possibility of pinning the opponent's starting point can result in shorter

than usual total plies in a game. However, when both players have pinned opponent checkers in their home board, the game lasts longer than usual (up to 150 plies) because of a large number of no-move plies.

Apart from longer sequences, Plakoto and Fevga variants have larger average branching factors. According to our experiments, the branching factor (in non-chance layers) of standard backgammon is 16[2], while for Fevga it is 25 and for Plakoto 23. This is mainly due to the fact that standard backgammon has fewer middle game positions where the number of available moves is at its peak.

2 Updating the Temporal Difference in Games

The theoretical background of reinforcement learning [11,12] and the TD(λ) [10] temporal difference learning algorithm that we used is described in [7]. This section explores alternatives for selecting the target of the TD updates and for the creation and updating of a self-play game sequence.

2.1 Determining the Target of the Update

In order to find the best move in a given situation, backgammon programs usually score each possible afterstate (that is the states resulting *after* the player has played a move) and select the move that produces the afterstate with the biggest score.

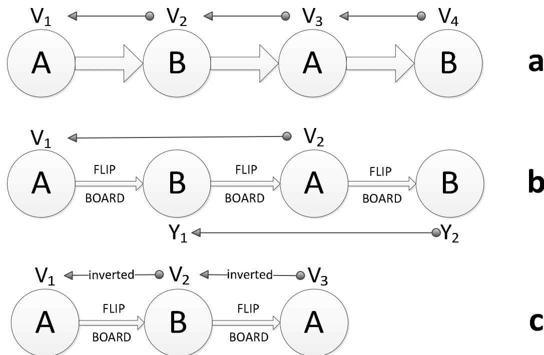


Fig. 2. Alternate updating methods of the temporal difference in two player zero-sum games. Method a: Update the values without flipping the board. Requires input(s) to designate which player is on the move. Method b: Updates are split into two. Method c: Updates are done on the inverted value of the next player. Circles indicate a position after a player (A or B) has made a move (afterstate).

An important implementation detail for a TD+NN learning system is the selection of input-target pairings for the TD update. In previous work we split up each training game into two training sequences, one for the afterstates of the first player and

[2] Several sources (e.g., [15]) claim a branching factor of 20 for standard backgammon. This number may have been calculated in conjunction with resignations and doubling cube drops.

another for the afterstates of the second player, and we updated these sequences separately (Fig. 2b). In this work we made one straightforward, yet effective improvement: instead of splitting up each training game into two, we keep one training sequence and we update each player's afterstate using as target the inverted value of the other's player afterstate on the next move (Fig. 2c). Both methods flip the board so as both players' afterstates are given to the neural network as if it is the first player to move. This is different from the approach used originally by TD-Gammon (Fig. 2a), where there was no flipping of the board and the neural network learned to play the game for both sides, identifying the side to move by two binary inputs. We believe the board-flipping approach has the potential of getting an improved performance as the expressiveness of the neural network is increased.

2.2 Sequence Creation and How to Update

Contrary to standard backgammon, when we started making programs for the variants Plakoto and Fevga, there was no other program close to expert play, nor were databases of games available. Therefore, self-play was for us the only option for creating a game sequence. We examined the following options for creating and updating a self-play game.

(1) Learning online (each update is done immediately after a move is played).
(2) Learning offline (updates are done incrementally after the game ends)
   (a) Forward offline: Updates are done starting from the first position of the game and ending at the terminal position.
   (b) Reverse offline: Updates are done starting from the terminal position of the game and ending at the first.
   (c) Reverse offline recalc: As previous, but recalculate target value after each update.

The intuition of updating backwards an offline game is that updates of non-terminal states will be more informed as the reward of the outcome of the game is received on the first update of the game. This is enhanced with the addition of the recalculation of the target value. Online updates have the benefit of learning while the game is in progress; however there is a chance that at the start of a training, where moves are more or less random, the agent will get stuck or progress slowly.

Preliminary experiments with all of the above methods showed that the slowest method was forward offline, particularly in the Fevga variant, with the others resulting in roughly the same performance (Fig. 3). The reverse offline method with recalculation of the target value learns faster than all others at the start of the training and continues to have a good performance afterwards. The downside is that more computation is needed in order to recalculate the target value at every step. However, this was not felt in our case since the creation of a game sequence is much more time consuming than the time to make the updates. Even with slower learning progress, all methods were found to reach the same level. So, whatever the final performance gains described later in the paper, they were only due to changing the updating method from (b) to (c) (2.1).

In our previous experiments, we used the forward offline method. Following the experiments mentioned in this section, all experiments in this paper were conducted using reverse offline with target recalculation.

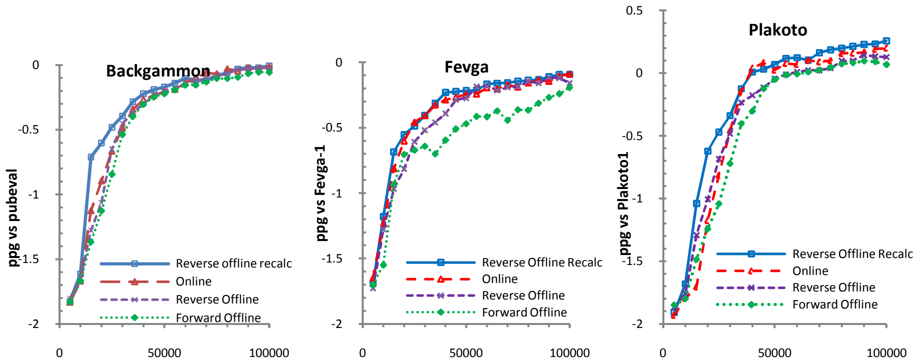


Fig. 3. Training progress of methods for sequence creation and update in Backgammon (left), Fevga (middle) and Plakoto (right). Every line is the average of 10 different training runs starting from the same random weights. For speed reasons, NNs in all games have 10 hidden units and no expert features. Benchmark opponents are pubeval for backgammon, Fevga-1 for Fevga, and Plakoto-1 for Plakoto.

3 Experimental Results

We compared the proposed training method to the previous one [7] by training again the NNs with the added expert features. For Plakoto, the new agent was named Plakoto3 and has exactly the same inputs as Plakoto2. For Fevga, we trained two new NNs: Fevga-4 has the same procedure as Fevga-2, whereas Fevga-5 has the same intermediate reward as Fevga-3. Table 2 shows all the techniques used by the various versions examined in this paper.

3.1 Results in the Plakoto and Fevga Variants

Earlier results [7] showed that Fevga-2's strategy was much different from the one considered by the human experts, even with features that recognized the presence of primes (six consecutive made points) in a position. To clarify the importance of primes more precisely, a new NN was trained (Fevga-3) where the agent learned with the same input units as Fevga-2, but with one important difference: when reaching a position with a prime formation, the target of the TD update was made a constant value instead of the next position value. This had as a result that the learned strategy was based on the creation of primes, which is roughly equivalent to what is perceived by experts as the best strategy. Results showed that the riskier strategy of Fevga-2 scores more points against the benchmark program Tavli3D [13] than Fevga-3, but when getting them to play against each other Fevga-2 was a little bit inferior. To preserve continuity with our previous work, we continued to benchmark our training progress with the open source program Tavli3D, which at the time of writing was the only open source program that can play these variants.

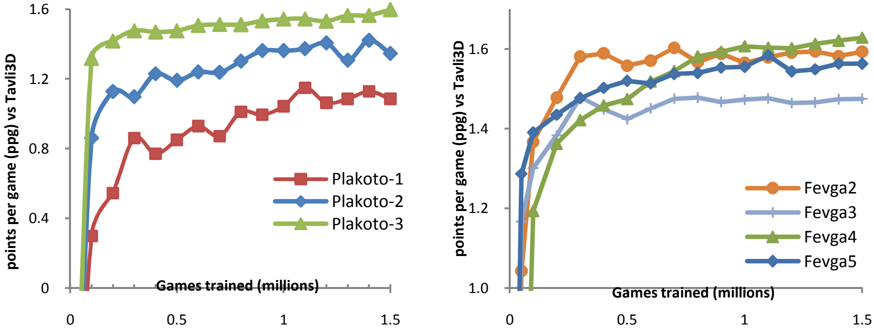


Fig. 4. Training progress of all trained NNs against the Tavli3D benchmark program in the Plakoto variant (**Left**) and the Fevga variant (**Right**)

All networks had 100 hidden neurons and were trained to 1.5 million games. For simplicity, we fixed the value of λ to zero for the experiments conducted in this paper. For λ>0 and reverse updates, care must be taken when taking future time steps into consideration: since every time step is viewed as the first player, any value taken by future time steps that is not a move by the player making the current update must be inverted. As the initial training of Fevga-2 and Fevga-3 were only 700,000 games, we extended their training (with the same initial λ=0.7) to match the new ones. During the training, we periodically saved the weights of each NN and we tested the networks against Tavli3D for 10,000 test games each, half as the first player and half as the second player (Fig.4). The result of the tested games sum up to the form of estimated *points per game* (ppg) and is calculated as the mean of the points won and lost.

We also tested the best set of weights of each NN by playing tournaments against each other at (1-ply) as well as by implementing a straightforward look-ahead procedure using the expectimax algorithm [5] at 2-ply depth (Table 3). In order to speed up the testing time, this expansion of depth-2 was performed only for the best 15 candidate moves (forward pruning). For the same reason, the total amount of testing games using 2-ply was reduced to 1,000 per test.

Table 2. Summary of techniques used by the various agents

Plakoto Agent	Updating method (Fig. 2)	Sequence creation and update direction	Fevga agent	Updating method (Fig. 2)	Sequence creation and update direction	Intermediate reward
Plakoto-1	b	Forward offline	**Fevga-2**	b	Forward offline	No
Plakoto-2	b	Forward offline	**Fevga-3**	b	Forward offline	Yes
Plakoto-3	c	Reverse offline recalc	**Fevga-4**	c	Reverse offline recalc	No
			Fevga-5	c	Reverse offline recalc	Yes

Table 3. Comparison of various agents at 1-ply and 2-ply for Plakoto (Left) and Fevga (Right). All results are in points per game (ppg) with respect to the player on the row. Players on columns always use 1-ply.

	Tavli3D	Plakoto1	Plakoto2
Plakoto-1	1-ply: +1.15 2-ply: +1.36	*	*
Plakoto-2	1-ply: +1.46 2-ply: +1.60	1-ply: +0.98 2-ply: +1.35	*
Plakoto-3	1-ply: +1.60 2-ply: +1.68	1-ply: +1.10 2-ply: +1.24	1-ply: +0.35 2-ply: +0.62

	Tavli3D	Fevga-2	Fevga-3	Fevga-4
Fevga-2	1-ply: +1.60 2-ply: +1.61	*	*	*
Fevga-3	1-ply: +1.52 2-ply: +1.53	1-ply: +0.03 2-ply: +0.49	*	*
Fevga-4	1-ply: +1.63 2-ply: +1.64	1-ply: +0.35 2-ply: +0.53	1-ply: +0.26 2-ply: +0.48	*
Fevga-5	1-ply: +1.58 2-ply: +1.59	1-ply: +0.42 2-ply: +0.60	1-ply: +0.32 2-ply: +0.45	1-ply: +0.02 2-ply: +0.14

Table 4. Analysis of some of the matches of Fevga-4 and Fevga-5

Match:	Fevga-5 vs Fevga-4		Fevga-4 vs Tavli3D		Fevga-5 vs Tavli3D	
	Fevga-5	**Fevga-4**	**Fevga-4**	**Tavli3D**	**Fevga-5**	**Tavli3D**
Single Wins	47.54%	39.52%	28.93%	2.74%	32.84%	2.86%
Double Wins	4.9%	8.04%	68.32%	0.01%	64.26%	0.04%
Total Wins	52.44%	47.56%	97.25%	2.75%	97%	3%
Final Score	+0.02ppg	-0.02ppg	+1.63ppg	-1.63ppg	+1.54ppg	-1.54ppg

The results in Plakoto show a significant increase in final performance. The performance of Plakoto-3 at 1-ply is equivalent to the performance of Plakoto-2 at 2-ply against Tavli3D. Additionally, Plakoto-3 learns faster than the other two agents.

In Fevga, Fevga-4 outperforms both Fevga-2 and Fevga-3 agents, while Fevga-5 outperforms all others except in the Tavli3D benchmark where it is inferior from Fevga-4, and Fevga-2 (Table 4). The explanation of this phenomenon is shown at Table 4. Against an inferior opponent Fevga-4 achieves more points because it wins more doubles due to its riskier strategy, while Fevga-5's safer strategy of building primes wins the same amount of games overall but fewer doubles. These new results show that the strategy learned by Fevga-4 and Fevga-5 is not different from the one learned from their previous counterparts (Fevga-2, and Fevga-3); obviously, the proposed training method learns the strategies better.

3.2 Feature Selection

An interesting observation was made while testing the strategies that were learned in the Plakoto variant. The resulting strategy was very conservative with regard to its starting point (also called the "mother" point). The agent correctly identified that it must not expose the last checker of its starting point, as it would potentially be open to a pinning attack that would automatically lose the maximum amount of points (double game). However, it could not discriminate the positions that such an attack could not be carried out by the opponent, and protected its first point even after we added the expert feature of pinning probability in Plakoto2 and Plakoto3 (Fig. 5). This resulted in obvious errors in a small number of positions. For example, the amount of equity lost for selecting the wrong move in Fig. 5 was calculated to 0.276ppg by making a 100,000 games rollout on each of the moves in question (Table 5).

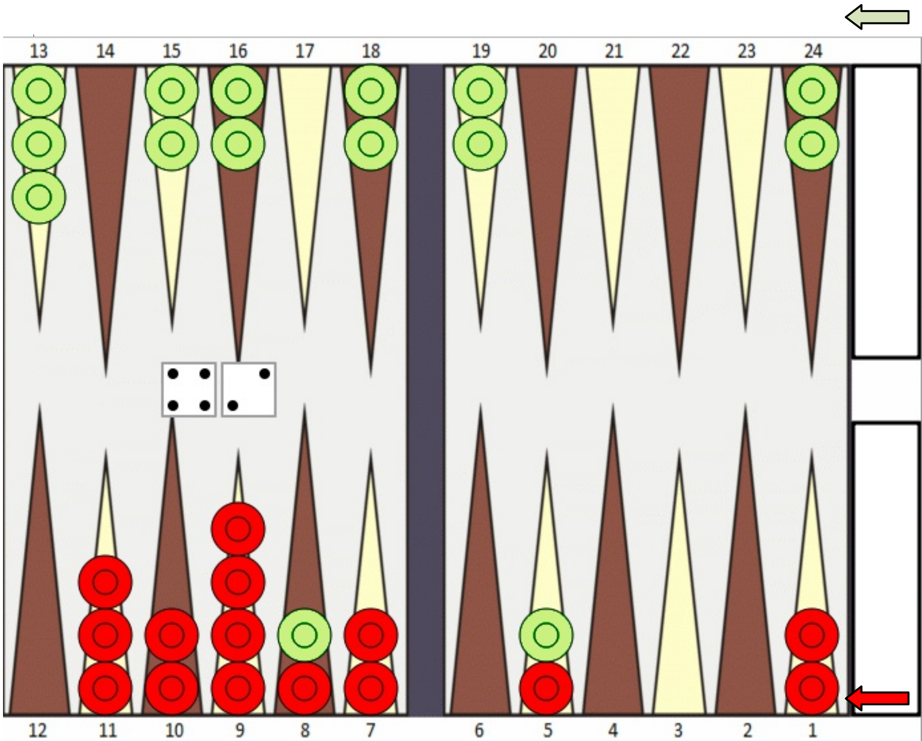


Fig. 5. Example of a position where agents Plakoto1-3 fail to produce the best move. The green player is to play roll 42. The best move here is 24/18 since the 24-point cannot be pinned by any dice roll. However, Plakoto1-3 agents prefer the clearly inferior move 24/20, 24/22 which give the opponent a pinning opportunity to get back into the game.

We suspected that the agent learned the harmful concept of leaving the first point open by the four raw features instead of the added expert feature "pinning probability at point 1". In order to confirm this we trained another agent (Plakoto-4) without the first of the four features for point 1, leaving only three features, one if 2 or more checkers are present, another if 3 or more checkers are present and a last one if 4 or more checkers are present. The resulting agent confirmed our suspicions, as it managed to learn the concept of "leaving the first point unprotected is bad" in a correct way, without committing the same mistakes of its predecessors. Evaluating Plakoto-4 final performance of 1.5 million trained games against Plakoto-3 in a 10,000 tournament resulted in equal performance. This may mean either (a) positions of this kind do not appear frequently and when they appear they did not seem to have a significant impact to the result, or (b) Plakoto-4 simply needs more training for the difference to tell.

Why was this concept not learned correctly by the other agents, especially when the other points where learned correctly? The concept of protecting the 1st point is one of the first things the agents learn, because it is the closest to the terminal position, the only position that receives reward, and because the random character of

the first self-play training games result in many "mother doubles". When confronted with two features to learn the concept, one being a binary input, and one a float input between 0 and 1, the neural network chooses the first one because it is the easier and the faster to learn. It would appear that the agent would have a chance to “unlearn” this later as learning progresses when the estimates of the NN are closer to the optimal. However, this is never done as these kind of positions appear rarely, because the agent has learned how (wrongly) to defend against.

Table 5. Evaluation and rollout analysis of the two best moves of the position in Fig.5. The first four columns show the evaluation of the Plakoto-3 and Plakoto-4 NNs after 1-ply and 2-ply look-ahead. The fifth and sixth column show the equity of the position by making a rollout analysis using Plakoto-3 and Plakoto-4. The last column shows the equity that was lost by selecting the inferior move. The equity loss was calculated on the average of the two rollouts.

Move	Plakoto-3 (1-ply) eval	Plakoto-3 (2-ply) eval	Plakoto-4 (1-ply) eval	Plakoto-4 (2-ply) eval	Rollout Plakoto-3	Rollout Plakoto-4	equity loss
24/22 24/20	1.020	1.048	0.942	1.046	0.983	0.968	0.276
24/18	0.692	1.082	1.140	1.248	1.259	1.243	-

4 Related Work

Temporal difference learning has been used for learning an evaluation function in most modern games. The KNIGHTCAP program [2] learned an evaluation function for chess using TD-Leaf, an extension to TD(λ) where updates are made not on the resulting positions of a training game, but on the leaf nodes of the principal variations resulting from alpha-beta searches from the previous and next positions. However, their approach only worked with initialization of the weights to a good starting point and could not learn from self-play. The rootstrap and treestrap algorithms introduced in [17] improved this approach by updating all interior nodes from the search tree towards the root node. Their program MEEP managed to achieve a rating of 2,338 Elo on Internet Chess Club for blitz games.

TD-Leaf was also tried in backgammon [3], but the authors could not improve the performance compared to an already trained NN with TD(λ). Following these results, we did not perform any experiments with TD-Leaf since the training time will have increased significantly.

In checkers [9], TD-Leaf was able to tune the weights of the best program of all time CHINOOK, to the same level of a set of weights previously manually tuned for a period of 5 years. In their approach, two separate set weights were trained, one for White and one for Black. The authors noted a similar performance for the two sets of weights, which indicates that a single set for both sides, as it was done with our approach, could be used.

In [18], a similar TD update like our proposed method was utilized for constructing a large architecture of several neural networks and for examining training using self-play and using an expert in the game of backgammon. In this work the update is called “minimax” update and the learning was conducted using the side of the first

player only. Results show that learning when playing against an expert is faster at first but ultimately reaches the same performance as self-play. This is explained by the analysis in section 2.2. Starting with expert training and continuing with self-play is usually a good hybrid approach.

5 Conclusion and Future Work

We have managed to increase the performance of our temporal difference learning architecture in the backgammon variants Plakoto and Fevga by making the target of the update the inverted value of the opponent's next state and by updating the game sequence starting from the terminal and working to the starting position. The problems found by learning overlapping features indicate that one must choose the features to be trained very carefully, or else risking suboptimal performance. An automatic process of selecting, comparing, and training the available features could be used in order to detect the beneficial from the problematic ones. This process, however, can be rather time consuming, especially when many games must be played for good learning (as is in backgammon) or the number of features is large (as is in chess, for example). These enhancements can be used in other games as well as in conjunction with other TD learning algorithms. As all our experiments were done with $\lambda=0$, an obvious continuation of this research is to determine if different values $\lambda>0$ can lead to improved performance.

We also plan to increase the number of backgammon variants that can be handled by *Palamedes*. Interesting candidates towards this direction are the acey-deucey, gioul and gul-bara variants. Our look-ahead procedure can be improved by searching in greater depths and by utilizing cutoff algorithms as in [4]. We are also planning to test *Palamedes* by participating in computer and human competitions.

Acknowledgements. This work was partially supported by the Greek State Scholarship Foundation (IKY). The authors would like to thank Anastasios Alexiadis for lending computing power for conducting some experiments presented in the paper, and the anonymous referees for their useful comments and suggestions.

References

1. BackGammon Variants, `http://www.bkgm.com/variants`
2. Baxter, J., Tridgell, A., Weaver, L.: Knightcap: a chess program that learns by combining td(lambda) with game-tree search. In: Shavlik, J.W. (ed.) Proc. 15th International Conf. on Machine Learning, pp. 28–36. Morgan Kaufmann, San Francisco (2001)
3. Baxter, J., Tridgell, A., Weaver, L.: Tdleaf(): Combining temporal difference learning with game-tree search. Australian Journal of Intelligent Information Processing Systems 5(1), 39–43 (1998)
4. Hauk, T., Buro, M., Schaeffer, J.: *-MINIMAX Performance in Backgammon. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 51–66. Springer, Heidelberg (2006)

5. Michie, D.: Game-playing and game-learning automata. In: Fox, L. (ed.) Advances in Programming and Non-Numerical Computation, pp. 183–200 (1966)
6. Palamedes, http://csse.uom.gr/~nikpapa/software.html
7. Papahristou, N., Refanidis, I.: Training Neural Networks to Play Backgammon Variants Using Reinforcement Learning. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A.I., Merelo, J.J., Neri, F., Preuss, M., Richter, H., Togelius, J., Yannakakis, G.N. (eds.) EvoApplications 2011, Part I. LNCS, vol. 6624, pp. 113–122. Springer, Heidelberg (2011)
8. Pubeval source code backgammon benchmark player, http://www.bkgm.com/rgb/rgb.cgi?view+610
9. Schaeffer, J., Hlynka, M., Vili, J.: Temporal Difference Learning Applied to a High-Performance Game-Playing Program. In: Proceedings IJCAI, pp. 529–534 (2001)
10. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine Learning 3(1), 9–44 (1988)
11. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Indroduction. MIT Press (1998)
12. Szepesvári, C.: Algorithms for Reinforcement Learning (Electronic Draft Version) (August 2010), http://www.sztaki.hu/~szcsaba/papers/RLAlgsInMDPs-lecture.pdf
13. Tavli3D, http://sourceforge.net/projects/tavli3d
14. Tesauro, G.: Practical issues in temporal differnce learning. Machine Learning 4, 257–277 (1992)
15. Tesauro, G.: Programming backgammon using self-teching neural nets. Artificial Intelligence 134, 181–199 (2002)
16. Tesauro, G.: Temporal Difference Learning and TD-Gammon. Communications of the ACM 38(3), 58–68 (1995)
17. Veness, J., Silver, D., Uther, W., Blair, A.: Bootstrapping from Game Tree Search. In: Advances in Neural Information Processing Systems, vol. 22, pp. 1937–1945 (2009)
18. Wiering, M.A.: Self-Play and Using an Expert to Learn to Play Backgammon with Temporal Difference Learning. Journal of Intelligent Learning Systems and Applications 2, 57–68 (2010)

CLOP: Confident Local Optimization for Noisy Black-Box Parameter Tuning

Rémi Coulom

Université de Lille, INRIA, CNRS, France

Abstract. Artificial intelligence in games often leads to the problem of parameter tuning. Some heuristics may have coefficients, and they should be tuned to maximize the win rate of the program. A possible approach is to build local quadratic models of the win rate as a function of program parameters. Many local regression algorithms have already been proposed for this task, but they are usually not sufficiently robust to deal automatically and efficiently with very noisy outputs and non-negative Hessians. The CLOP principle, which stands for Confident Local OPtimization, is a new approach to local regression that overcomes all these problems in a straightforward and efficient way. CLOP discards samples of which the estimated value is confidently inferior to the mean of all samples. Experiments demonstrate that, when the function to be optimized is smooth, this method outperforms all other tested algorithms.

1 Introduction

Authors of programs that play games, such as chess or Go, are faced with the problem of optimizing parameters. A game-playing program relies on heuristics for position evaluation, search-tree pruning, or time management. Most of these heuristics have parameters, and tuning them may improve the program's strength.

When optimizing parameters, a major difficulty is measuring the strength. The most usual approach is quite costly: it is based on the win rate against a reference opponent. In order to obtain an accurate measurement, it is necessary to play many games, which takes a large amount of computation time.

Since it is so costly, authors of game-playing programs sometimes try to avoid measuring win rates. Playing games may be replaced by testing over a database of one-move problems. It may also be replaced by machine-learning algorithms, such as temporal-difference methods [27,30].

Tuning parameters without measuring win rates may sometimes work, but it is dangerous; particularly, in the sense that it does not guarantee an optimal probability of winning. So far, it has not been proved that optimizing a criterion such as temporal difference also maximizes the strength.

Besides having no guarantee in terms of strength optimization, many learning algorithms also have a limited scope. For instance, temporal-difference methods can tune an evaluation function, but not selectivity or time management.

For a reliable and generic parameter-optimization method, it is necessary to measure the strength by the outcome of games played. The challenge is to come as close as possible to the optimal win rate with as few games as possible.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 146–157, 2012.
© Springer-Verlag Berlin Heidelberg 2012

1.1 Problem Definition

More formally, the problem addressed in this paper is the estimation of an optimal value $\boldsymbol{x}^*$ of a vector of n continuous bounded parameters $\boldsymbol{x} \in [-1, 1]^n$. Performance of a vector of parameters $\boldsymbol{x}$ is measured by its probability of success $f(\boldsymbol{x}) \in [0, 1]$. The value of $f(\boldsymbol{x})$ is not known, but can be estimated by observing the outcome of independent Bernoulli trials that succeed with probability $f(\boldsymbol{x})$. These trials are observed in sequence. For each trial, parameters can be chosen in the $[-1, 1]^n$ domain. After each trial, the optimization algorithm recommends a vector of parameters $\tilde{\boldsymbol{x}}$. The performance of the algorithm is measured by the simple regret $f(\boldsymbol{x}^*) - f(\tilde{\boldsymbol{x}})$. The objective is to find an algorithm that will make the simple regret go to zero as fast as possible.

The notion of "simple regret" is named in opposition to the more usual "online regret" of continuum-armed bandits algorithms [1,10]. When tuning a game-playing program, losing games does not matter. The objective is to optimize the final strength of the program, regardless of losses suffered while training.

The function f to be optimized will be assumed to have no local optima. The main difficulty that CLOP addresses is not getting out of tricky local optima, but dealing with noise.

1.2 Noisy Optimization

Because it has so many important applications in engineering, the problem of optimizing continuous parameters from noisy observations has been well studied. Many algorithms have been already proposed, even for the special case of binary response.

One of the oldest methods for optimization is stochastic gradient ascent. The principle of this approach is based on collecting samples of the function around the current parameters. These samples are used to estimate the gradient of the function. Parameters are then modified with a small step in the direction of the noisy gradient estimate. The most primitive form of this idea is the Kiefer-Wolfowitz algorithm [20]. The idea of Kiefer and Wolfowitz was improved in the multivariate case with the SPSA algorithm [28]. Several second-order refinements of SPSA were proposed [23,29].

A second kind of approach is population-based algorithms, such as evolution strategies or genetic algorithms [8,16,15]. These methods operate over a set of points in a parameter space, called the *population*. They iterate the following process: first, evaluate all elements of the population; then, discard those that perform badly; then, generate new elements similar to those that perform well.

Yet, a third approach is simulated annealing [21]. Simulated annealing is often used for combinatorial optimization with noiseless performance measurements. But it was generalized to the optimization of noisy functions [5], and to the optimization of continuous parameters [24].

1.3 Using Response-Surface Models

The algorithms presented so far often have to make a trade-off between fast inaccurate evaluation of many different parameter values, and slow accurate evaluation of few different parameter values. This trade-off is usually tuned by a meta-parameter that indicates the number of samples that are collected for each parameter value. In order to reduce noise, it may be necessary to replicate many independent measurements at the same point.

Many of these algorithms are also incremental, and discard old data. This may lead to a non-optimal use of all available information.

An approach that does not forget data and requires no trade-off between fast and accurate evaluation is the response-surface methodology [4]. The response-surface methodology fits a model to data, and performs optimization on the model. With response-surface models, it is not necessary to run more than one trial for each parameter value, which allows a dense coverage of the parameter space.

An important question when optimizing with response-surface models is the choice of a model. A first approach is to choose non-parametric models that are sufficiently general to fit any function [26,19,2,17,31,12,18]. A second approach is to use simpler models (linear or quadratic), over a shrinking domain [13,25,11,7]. The first approach is able to find the global optimum of a function with many local optima, whereas the second approach only performs local optimization.

Existing algorithms based on local regression all have some major limitations. The traditional response-surface methodology is not completely automated, and requires human judgment. Q2 [25] is rather similar to CLOP, except for its criterion for shrinking the region of interest (ROI): a sample is discarded only if there is a good confidence that the maximum of the quadratic regression lies within the ROI. This does not work if the Hessian is not definite negative, which happens frequently in practice (for instance, if one parameter turns out to have no influence on the strength). Noisy UOBYQA [11] uses a similar criterion, and it has the same defect. In addition, noisy UOBYQA takes the next sample at the estimated location of the optimum, which was found to be a rather inefficient sampling policy [25]. The trust-region method Elster and Neumaier [13] does not work in the very noisy case because it estimates the best parameters as those that obtained the best result so far. STRONG [7] has a proof of convergence, but (1) experiments show poor empirical performance compared to a straightforward stochastic gradient, and (2) the algorithm is extremely complicated.

The algorithm presented in this paper can deal in a simple, automatic and robust way with very noisy observations and a non-negative Hessian. The main difference with previous algorithms is in its criterion for deciding when to stop shrinking the regression area: the worst sample is discarded if its estimated value according to the regression is inferior with some level of confidence to the mean of all the remaining samples. Section 2 gives a detailed description of the algorithm and an intuitive analysis of its asymptotic rate of convergence. Section 3 presents empirical data that demonstrate its good performance compared to many alternative algorithms.

2 Algorithm

The general idea of the optimization algorithm is to perform regression over all the data, and then removing the worst samples (according to the regression) as long as sufficient samples are left. Samples are not removed one by one, because it would be too inefficient when the number of samples is high. Instead, a weight function, w, is computed with a formula that gives a weight close to zero to samples of which the estimated strength is confidently inferior to the average strength of samples. This process is iterated until convergence. Convergence is guaranteed by not allowing a weight to increase.

2.1 Detailed Algorithm Description

The details of the optimization algorithm for quadratic regression are given in Algorithm 1. The first parameter of the function is a positive number H that indicates how local the regression will be. $(\boldsymbol{x}_i, y_i)$ are pairs of past inputs and their outputs. y_i may be either 0 (loss) or 1 (win). QuadraticLogisticRegression is a function that performs weighted quadratic logistic regression, that is to say $f(\boldsymbol{x})$ is approximated at iteration k by $1/\big(1 + e^{-q_k(\boldsymbol{x})}\big)$, where q_k is a quadratic function. LogisticMean is logistic regression by a constant. Both QuadraticLogisticRegression and LogisticMean compute the maximum a posteriori with a Gaussian prior of variance 100. ConfidenceDeviation is the standard deviation of the posterior of LogisticMean.

The next sample is chosen at random (using Gibbs sampling) by using w as a probability density. The theory of optimal design offers many alternatives [6,14], but sampling according to w outperformed them in experiments. A problem with

Algorithm 1. Quadratic CLOP

procedure QuadraticCLOP($H, \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N$)
  $w_0 \leftarrow \lambda\boldsymbol{x}.1$ ▷ a function of $\boldsymbol{x}$ that returns 1
  $W_0 \leftarrow N$
  $k \leftarrow 0$

  repeat
    $w \leftarrow \lambda\boldsymbol{x}.\min_{i=0}^{k} w_i(\boldsymbol{x})$ ▷ weight function
    $k \leftarrow k + 1$
    $q_k \leftarrow$ QuadraticLogisticRegression($w, \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N$)
    $\mu_k \leftarrow$ LogisticMean($w, \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N$)
    $\sigma_k \leftarrow$ ConfidenceDeviation($w, \boldsymbol{x}_1, y_1, \ldots, \boldsymbol{x}_N, y_N$)
    $w_k \leftarrow \lambda\boldsymbol{x}.e^{(q_k(\boldsymbol{x})-\mu_k)/(H\sigma_k)}$
    $W_k \leftarrow \Sigma_{i=1}^{N} \min\big(w(\boldsymbol{x}_i), w_k(\boldsymbol{x}_i)\big)$
  until $W_k > 0.99 \times W_{k-1}$

  $\boldsymbol{x}_{N+1} \leftarrow$ Random(w) ▷ next sample, distributed like w
  $\tilde{\boldsymbol{x}} \leftarrow \Sigma_{i=1}^{N+1} w(\boldsymbol{x}_i)\boldsymbol{x}_i / \Sigma_{i=1}^{N+1} w(\boldsymbol{x}_i)$ ▷ estimated optimal
end procedure

optimal design is that it assumes that the model perfectly fits the data, which is almost always wrong in practice. Even when lack of fit is taken into consideration [32], these algorithms take samples near the edge of the sampling domain. Samples near the edge are rapidly discarded when the domain is shrinking. It might be possible to do better, but sampling according to w is straightforward and works well.

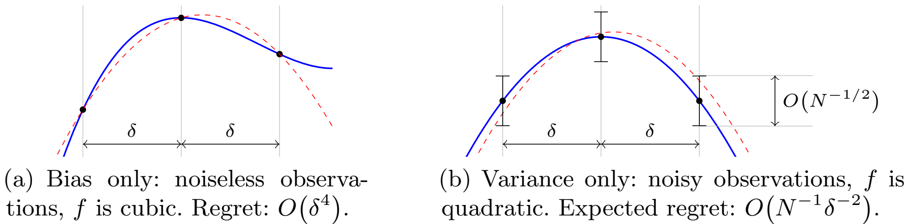
For quadratic regression, the maximum is estimated by the weighted average of samples. A different possibility would be to maximize the quadratic regression, but the estimation of the weighted average turned out to be more robust and perform better. In particular, it works even if the regression is not definite negative.

2.2 Choice of H and Asymptotic Rate of Convergence

The most important meta-parameter of this algorithm is H. Besides H, other parameters are the priors of regressions, and the 0.99 constant that decides the end of the loop. But they have actually little influence on the performance. H tunes the bias-variance trade-off by making regression more or less local, and should be chosen carefully.

Figure 1 shows an analysis of the asymptotic bias-variance trade-off. In Algorithm 1, the "height" of the local regression is $H\sigma_k$. It should be proportional to the square of the "width" δ. If we assume that $\sigma_k = O\big(N^{-1/2}\big)$, this means that the optimal asymptotic rate of convergence is obtained with $H = O(N^{1/6})$.

An attempt at finding a more precise optimal value for H [3] shows that it depends on the magnitude of the cubic term: if the function to be optimized is perfectly quadratic, then $H = \infty$ is optimal. Otherwise, H should be smaller. Also, in the small-sample case, terms of degree four or more might not be negligible. So, it is difficult to find the optimal value of H. But, as will be shown in Section 3.1, choosing a constant value of $H = 3$ works quite well in practice, in a really wide range of situations.



(a) Bias only: noiseless observations, f is cubic. Regret: $O\big(\delta^4\big)$.

(b) Variance only: noisy observations, f is quadratic. Expected regret: $O\big(N^{-1}\delta^{-2}\big)$.

Fig. 1. The figures (a) and (b) illustrate an intuitive derivation of the optimal asymptotic bias-variance trade-off for local quadratic regression in dimension one [3]. Assuming N observations are made at x^*, $x^*-\delta$, and $x^*+\delta$, it is possible to calculate expected simple regret in both situations. The optimal trade-off is when they are the same, which gives $\delta = O\big(N^{-1/6}\big)$, and a simple regret of $O\big(N^{-2/3}\big)$. It was proved that $O\big(N^{-2/3}\big)$ is optimal when optimizing functions with bounded third-order derivatives [9], that is to say no algorithm can do better.

It is worth noting that the principle of CLOP is universal, and can be applied to any kind of regression, not only quadratic. So it is possible to use cubic regression or any other arbitrary polynomial regression instead. Such an approach may reach the optimal bound by Chen [9], that is to say $O\big(N^{-s/(s+1)}\big)$ simple regret for polynomial regression of degree s, provided f is sufficiently smooth. Figure 5 shows an experiment where cubic regression does outperform the best possible quadratic regression.

3 Experiments

The performance of CLOP was measured by artificial problems. Table 1, Fig. 2, and Fig. 3 show the artificial functions that were optimized. When an exponent is added to a problem name, it means that the dimension is multiplied by this exponent, and $r(\boldsymbol{x})$ is the average of $r(\boldsymbol{x})$ for each dimension (see Fig. 3 for the example of LOG^2). The source code of the program that produced these results is available at `http://remi.coulom.free.fr/CLOP/`.

Table 1. Problem definitions. $f(\boldsymbol{x}) = 1/\big(1+e^{-r(\boldsymbol{x})}\big)$. $\boldsymbol{x} \in [-1,1]^n$.

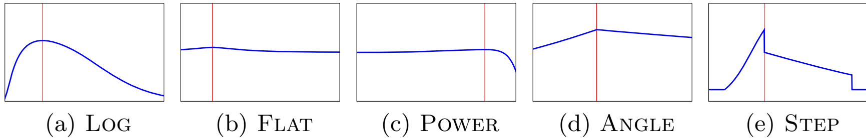
LOG	$n=1$	$r(x) = 2\log(4x+4.1) - 4x - 3$
FLAT	$n=1$	$r(x) = 0.2/\big(1+6(x+0.6)^2+(x+0.6)^3\big)$
POWER	$n=1$	$r(x) = 0.05(x+1)^2 - ((x+1)/2)^{20}$
ANGLE	$n=1$	$r(x) = 1 + \begin{cases} \sqrt{2} - 2\sqrt{0.3-x} & \text{if } x < -0.2, \\ \sqrt{2} - \sqrt{x+2.2} & \text{otherwise.} \end{cases}$
STEP	$n=1$	$r(x) = \begin{cases} -2 & \text{if } x < -0.8, \\ -2+6(x+0.8) & \text{if } -0.8 < x < -0.3, \\ -(x+0.3)/1.1 & \text{if } -0.3 < x < 0.8, \\ -2 & \text{otherwise.} \end{cases}$
ROSENBROCK	$n=2$	$r(\boldsymbol{x}) = 1.0 - 0.1\big((1-a)^2 + (b-a^2)^2\big)$, $a = 4x_1$, $b = 10x_2 + 4$
CORRELATED	$n=2$	$r(\boldsymbol{x}) = 0.2(g(10(x_1+x_2+0.1)) + g(x_1-x_2+0.9)) + 0.2$, with $g(x) = -x^4 + x^3 - x^2$

3.1 Effect of Meta-parameter *H*

Figure 4 shows the effect of H when optimizing three different functions. As predicted in section 2.2, the optimal value of H depends on the quadraticity of the function to be optimized. For a moderately non-quadratic function such as LOG, higher values of H perform better than for a strongly non-quadratic function such as POWER. Results confirm the $O\big(N^{-2/3}\big)$ asymptotic simple regret with $H = O\big(N^{1/6}\big)$. Although it is not clear how to find the optimal value of H in practice, using a constant value of $H = 3$ seems to perform well in all cases.

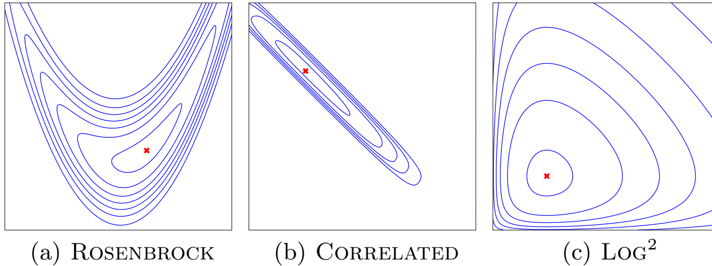
3.2 Comparison with Other Algorithms

Figures 5 and 6 compares CLOP to many algorithms. CLOP is best for smooth functions, works similarly for ANGLE, and does not work well for STEP.



(a) Log (b) Flat (c) Power (d) Angle (e) Step

Fig. 2. Plots of one-dimensional problems. x^* is indicated by a vertical line.



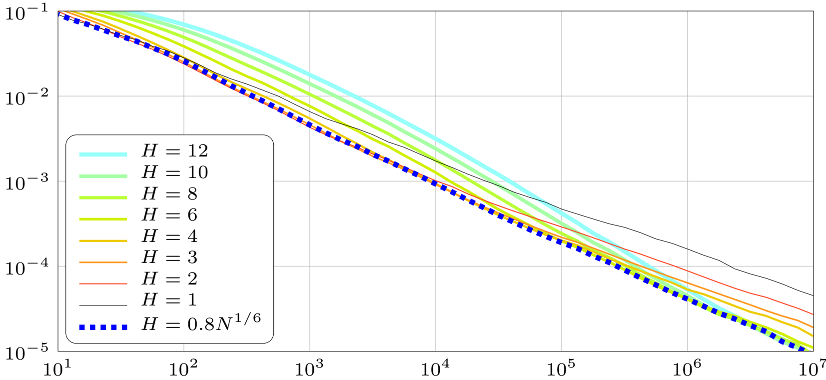
(a) Rosenbrock (b) Correlated (c) Log^2

Fig. 3. Plots of two-dimensional problems. Lines of constant probability are plotted every 0.1. $\boldsymbol{x}^*$ is indicated by a cross.

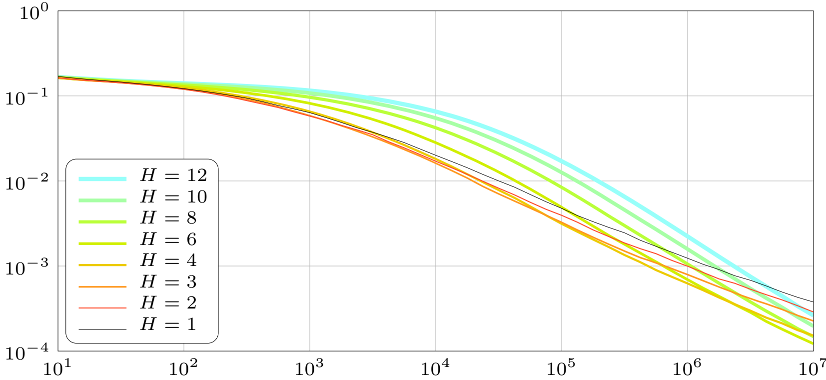
The population-based algorithms that were tested are variations of the Cross-Entropy method (CEM), using an independent Gaussian distribution. The basic version [8] was tested, as well as a version improved by dynamic parameter smoothing [16]. Initial population distribution was uniform random over $[-1, 1]^n$. Meta-parameters of the algorithms were: population = 100, elite = 10, initial batch size = 10, batch-size growth rate = 1.15, smoothing = 0.9, dynamic smoothing (improved version only) = 0.1. UH-CMA-ES [15] was tested too, but it was clearly not designed for that kind of problem, and it does not work well. Results were not plotted, because it sometimes fails to remain within the $[-1, 1]$ interval, even when started at $x = 0$ with a small variance.

A second algorithm that was tested is UCT [22], applied to a recursive binary partitioning of the parameter space. $\tilde{\boldsymbol{x}}$ is determined by choosing the child with the highest win rate.

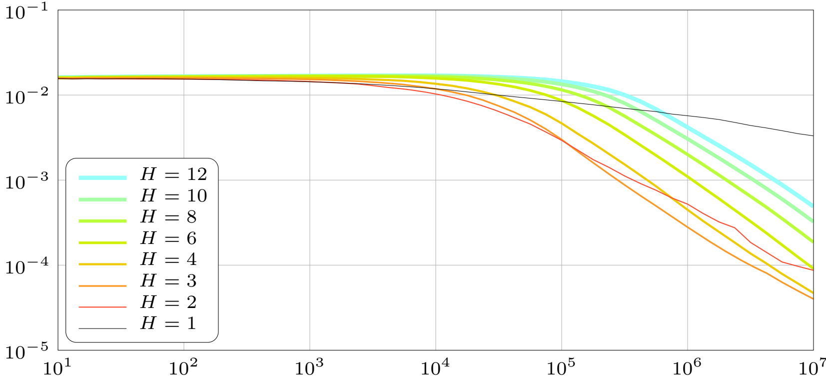
Finally, the SPSA algorithm [28] was tested. SPSA* is plain SPSA, with manually optimized meta-parameters ($a = 3$, $A = 0$, $\alpha = 1$, $c = 0.1$, $\gamma = 1/6$) starting at $\theta_0 = 0$. It performed quite well for Log, reaching the $O\big(N^{-2/3}\big)$ optimal asymptotic rate of convergence. But the performance of SPSA is rather sensitive to a good choice of meta-parameters, and these values do not work in practice for other problems. Many adaptive forms of SPSA have been proposed to automatically tune meta-parameters. RSPSA [23] is one such algorithm. Its meta-parameters were manually chosen to minimize simple regret at 10^5 samples (batch size = 1000, $\eta_+ = 1$, $\eta_- = 0.9$, $\delta_0 = 0.019$, $\delta_- = 0$, $\delta_+ = 0.02$, $\rho = 25$). These meta-parameters clearly overfit the problem, but they still do not outperform CLOP. Enhanced Adaptive SPSA (E2SPSA [29]) is another form of SPSA with better convergence guarantees. E2SPSA was not tested, but it can be expected that it would be at least twice slower than SPSA*, because half



(a) Log



(b) Log^5



(c) Power

Fig. 4. Effect of meta-parameter H. Simple regret (averaged over 1,000 replications) is plotted as a function of the number of samples.

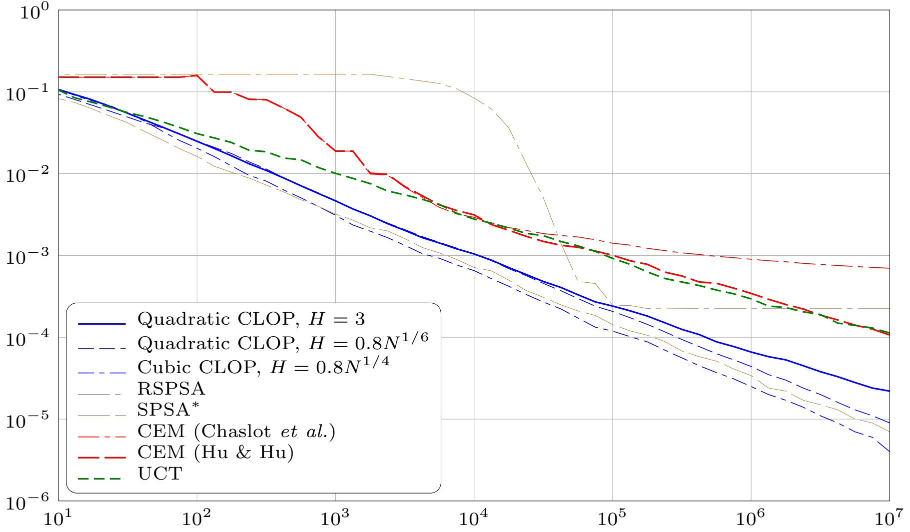


Fig. 5. Comparison of many algorithms applied to the LOG problem

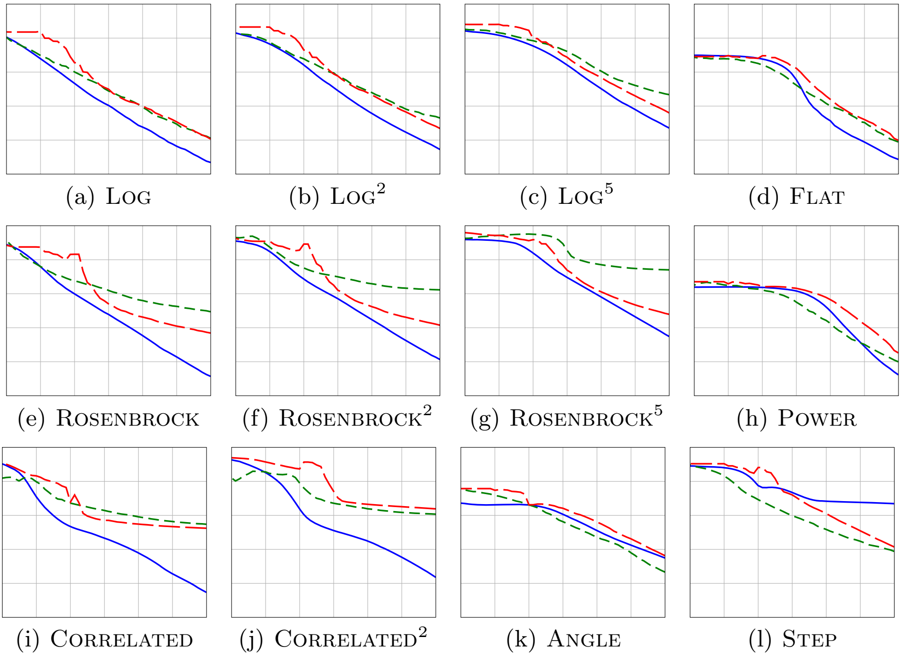


Fig. 6. Many problems. Scale: regret from 10^{-5} to 1, samples from 10 to 10^7. Legend: ——Quadratic CLOP ($H = 3$), - - - UCT, — —CEM (Hu & Hu).

of the samples it collects are not used for gradient estimation. Also, E2SPSA only adapts a, but not c. So, it probably could not work well in problems such as CORRELATED. Still, the good performance of SPSA* is a clear sign that adaptive forms of SPSA could approach the performance of CLOP.

An aspect worth mentioning is the computational cost of these algorithms. Because logistic regression is a costly operation, CLOP tends to be slower. However, it can be made fast with a few tricks. First, w does not have to be re-computed at every sample: when the regression has been computed by N samples, $1 + N/10$ samples are collected without updating the regression. Second, an additional speed-up can be obtained by taking more than one sample at the same location. Experiments were run by averaging 1,000 runs of 10^7 samples, on a 24-core PC. For the LOG problem, CEM takes 1'20", CLOP 9'57", and UCT 21'19". UCT is slow because it was not particularly optimized, but it could certainly be sped-up considerably by using replications, too. Anyway, the cost of these algorithms is negligible compared to the cost of playing even super-fast games.

4 Conclusion

In summary, CLOP is a new approach to black-box optimization with local response-surface models. CLOP is completely automated, robust to very noisy outputs, and to non-negative Hessians. The algorithm is straightforward, and has only one meta parameter, H, that does not have a critical influence on performance. In practice, using a constant value of $H = 3$ works quite well in a wide range of function shapes and experiment sizes. Experiments demonstrate the excellent performance of CLOP for optimizing smooth functions.

In the future, CLOP could be applied to less noisy problems. It might even be possible to make it work efficiently for completely noiseless black-box optimization. This would probably require low-discrepancy algorithms (like in Q2 [25]), rather than random sampling. A second interesting question is the application of CLOP to other forms of regression. Quadratic regression is the most obvious and popular approach for local optimization, but, as was demonstrated in experiments, using more complex forms of regression, such as cubic regression, might produce better results. Finally, although the CLOP algorithm turned out to be extremely reliable in experiments, it would be good to have a mathematical proof of its convergence.

Acknowledgments. The work was supported in part by the IST Programme of the European Community, under the PASCAL2 Network of Excellence, IST-2007-216886. Moreover, it was supported in part by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the "CPER 2007–2013". This publication reflects only the author's views.

References

1. Agrawal, R.: The continuum-armed bandit problem. SIAM Journal on Control and Optimization 33(6), 1926–1951 (1995)
2. Anderson, B.S., Moore, A.W., Cohn, D.: A nonparametric approach to noisy and costly optimization. In: Langley, P. (ed.) Proceedings of the Seventeenth International Conference on Machine Learning, pp. 17–24. Morgan Kaufmann (2000)
3. Boesch, E.: Minimizing the mean of a random variable with one real parameter (2010)
4. Box, G.E.P., Wilson, K.B.: On the experimental attainment of optimum conditions (with discussion). Journal of the Royal Statistical Society 13(1), 1–45 (1951)
5. Branke, J., Meisel, S., Schmidt, C.: Simulated annealing in the presence of noise. Journal of Heuristics 14, 627–654 (2008)
6. Chaloner, K.: Bayesian design for estimating the turning point of a quadratic regression. Communications in Statistics—Theory and Methods 18(4), 1385–1400 (1989)
7. Chang, K.H., Hong, L.J., Wan, H.: Stochastic trust region gradient-free method (STRONG)—a new response-surface-based algorithm in simulation optimization. In: Henderson, S.G., Biller, B., Hsieh, M.H., Shortle, J., Tew, J.D., Barton, R.R. (eds.) Proceedings of the 2007 Winter Simulation Conference, pp. 346–354 (2007)
8. Chaslot, G.M.J.B., Winands, M.H.M., Szita, I., van den Herik, H.J.: Cross-entropy for Monte-Carlo tree search. ICGA Journal 31(3), 145–156 (2008)
9. Chen, H.: Lower rate of convergence for locating a maximum of a function. The Annals of Statistics 16(3), 1330–1334 (1988)
10. Coquelin, P.A., Munos, R.: Bandit algorithms for tree search. In: Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (2007)
11. Deng, G., Ferris, M.C.: Adaptation of the UOBYQA algorithm for noisy functions. In: Perrone, L.F., Wieland, F.P., Liu, J., Lawson, B.G., Nicol, D.M., Fujimoto, R.M. (eds.) Proceedings of the 2006 Winter Simulation Conference, pp. 312–319 (2006)
12. Deng, G., Ferris, M.C.: Extension of the DIRECT optimization algorithm for noisy functions. In: Henderson, S.G., Biller, B., Hsieh, M.H., Shortle, J., Tew, J.D., Barton, R.R. (eds.) Proceedings of the 2007 Winter Simulation Conference, pp. 497–504 (2007)
13. Elster, C., Neumaier, A.: A method of trust region type for minimizing noisy functions. Computing 58(1), 31–46 (1997)
14. Fackle Fornius, E.: Optimal Design of Experiments for the Quadratic Logistic Model. Ph.D. thesis, Department of Statistics, Stockholm University (2008)
15. Hansen, N., Niederberger, A.S.P., Guzzella, L., Koumoutsakos, P.: A method for handling uncertainty in evolutionary optimization with an application to feedback control of combustion. IEEE Transactions on Evolutionary Computation 13(1), 180–197 (2009)
16. Hu, J., Hu, P.: On the performance of the cross-entropy method. In: Rossetti, M.D., Hill, R.R., Johansson, B., Dunkin, A., Ingalls, R.G. (eds.) Proceedings of the 2009 Winter Simulation Conference, pp. 459–468 (2009)
17. Huang, D., Allen, T.T., Notz, W.I., Zeng, N.: Global optimization of stochastic black-box systems via sequential kriging meta-models. Journal of Global Optimization 34(3), 441–466 (2006)

18. Hutter, F., Bartz-Beielstein, T., Hoos, H., Leyton-Brown, K., Murphy, K.: Sequential model-based parameter optimisation: an experimental investigation of automated and interactive approaches. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuß, M. (eds.) Empirical Methods for the Analysis of Optimization Algorithms, ch.15, pp. 361–411. Springer (2010)
19. Jones, D.R., Schonlau, M., Welch, W.J.: Efficient global optimization of expensive black-box functions. Journal of Global Optimization 13, 455–492 (1998)
20. Kiefer, J., Wolfowitz, J.: Stochastic estimation of the maximum of a regression function. Annals of Mathematical Statistics 23(3), 462–466 (1952)
21. Kirkpatrick, S., Gelatt Jr., C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
22. 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)
23. Kocsis, L., Szepesvári, C.: Universal parameter optimisation in games based on SPSA. Machine Learning 63(3), 249–286 (2006)
24. Locatelli, M.: Simulated annealing algorithms for continuous global optimization. In: Handbook of Global Optimization II, pp. 179–230. Kluwer Academic Publishers (2002)
25. Moore, A.W., Schneider, J.G., Boyan, J.A., Lee, M.S.: Q2: Memory-based active learning for optimizing noisy continuous functions. In: Shavlik, J. (ed.) Proceedings of the Fifteenth International Conference of Machine Learning, pp. 386–394. Morgan Kaufmann (1998)
26. Salganicoff, M., Ungar, L.H.: Active exploration and learning in real-valued spaces using multi-armed bandit allocation indices. In: Prieditis, A., Russell, S.J. (eds.) Proceedings of the Twelfth International Conference on Machine Learning, pp. 480–487. Morgan Kaufmann (1995)
27. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal 3(3), 210–229 (1959)
28. Spall, J.C.: Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. IEEE Transactions on Automatic Control 37, 332–341 (1992)
29. Spall, J.C.: Feedback and weighting mechanisms for improving Jacobian estimates in the adaptive simultaneous perturbation algorithm. IEEE Transactions on Automatic Control 54(6), 1216–1229 (2009)
30. Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM 38(3), 58–68 (1995)
31. Villemonteix, J., Vazquez, E., Walter, E.: An informational approach to the global optimization of expensive-to-evaluate functions. Journal of Global Optimization (September 2008)
32. Wiens, D.P.: Robustness of design for the testing of lack of fit and for estimation in binary response models. Computational Statistics & Data Analysis 54(12), 3371–3378 (2010)

Analysis of Evaluation-Function Learning by Comparison of Sibling Nodes

Tomoyuki Kaneko[1] and Kunihito Hoki[2]

[1] Department of Graphics and Computer Sciences, The University of Tokyo
[2] Department of Communication Engineering and Informatics, The University of Electro-Communications

Abstract. This paper discusses gradients of search values with a parameter vector θ in an evaluation function. Recent learning methods for evaluation functions in computer shogi are based on minimization of an objective function with search results. The gradients of the evaluation function at the leaf position of a principal variation (PV) are used to make an easy substitution of the gradients of the search result. By analyzing the variations of the min-max value, we show (1) when the min-max value is partially differentiable and (2) how the substitution may introduce errors. Experiments on a shogi program with about a million parameters show how frequently such errors occur, as well as how effective the substitutions for parameter tuning are in practice.

1 Introduction

An evaluation function is a function that estimates how preferable a position is for a max player and is an essential part of a heuristic search in a two-player perfect information game. Machine learning of evaluation functions is an important topic in artificial intelligence research [7]. For many years, the best shogi playing programs [11] used hand-tuned evaluation functions; however, as of 2010, almost all of the strongest programs have incorporated machine learning techniques for tuning the parameters in their evaluation functions. The main idea behind their learning methods and other methods [13,6,1,14,16] is to make the search results with their evaluation function agree with training data consisting of grandmasters' moves. In other words, the goal of adjusting the parameters is to make the min-max value for the grandmasters' move higher than that for any other legal move of each position in the training data. A min-max value is the result of min-max search defined in Section 3.

To update the parameters in an evaluation function efficiently, one needs to know how a min-max value will change with the parameters. However, a min-max value is not always partially differentiable with respect to one of these parameters. Accordingly, the gradients at the leaf position in a principal variation (PV) have been used as a substitute for the gradients of the min-max value. This has been done in studies related to ours [14,16,9] and in TDLeaf [2]. Although the substitution is intuitive, we show that the partial derivative of the leaf position in a PV does not always equal the partial derivative of the min-max value even

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 158–169, 2012.
© Springer-Verlag Berlin Heidelberg 2012

if the min-max value is partially differentiable. More precisely, the variation of a min-max value is, in general, determined by the gradients of the two special leaves depending on the search tree. Experiments on a shogi program with about a million parameters evaluated the accuracy of the gradient approximation using the leaf positions. We also discuss their effectiveness for controlling the search results when the parameters are stored in integral type.

2 Related Work

Backgammon [17] and Othello [4] are well-known successes regarding learning of evaluation functions in game programming. In contrast, it is interesting that chess programs with evaluation functions tuned by machine learning have not outperformed programs with hand-tuned evaluation functions [2,18]. For instance, it has been reported that the large majority of the features and weights in the Deep Blue evaluation function were created/tuned by hand [5].

Bonanza is the first shogi program that won the CSA championship with evaluation functions fully tuned by machine learning in 2006 [9]. Before this pioneering work, shogi programs yielded by learning remained very weak despite the efforts by the shogi programmers and researchers (e.g., [3]). Machine learning of evaluation functions quickly became popular afterwards, to the point that, today, almost all strong programs use a similar learning method.[1] The idea of comparing a grandmaster's move with other legal moves [9] has already been exploited in computer chess and other games [13,6,1,14,16]. It is beyond the scope of this paper to discuss why the recent method [9] outperformed hand-tuned evaluation functions while other existing methods did not. One possibility, though, is that the method [9] has a high scalability. It is capable of dealing with millions of parameters and is a well-modeled optimization problem with a clear loss function and careful constraint terms.

3 Learning by Comparison of Moves

3.1 Objective Function to Be Minimized

Let us begin with a straightforward but intuitive goal that is to make the result of a one-ply search agree with the move selection of a grandmaster. In a one-ply search, the move with the highest evaluation value is selected. Thus, in order to meet our goal, a parameter vector in an evaluation function should be modified so that the grandmaster's move has the highest value among all the legal moves.

Let $\mathrm{eval}(p, \theta)$ be an evaluation value on a position p yielded by an evaluation function with a parameter vector θ. Part of our goal is represented by a constraint $\mathrm{eval}(p.m, \theta) - \mathrm{eval}(p.g_p, \theta) < 0$, where g_p is a grandmaster's move in position p and m is another legal move in p. Here, $p.m$ denotes the position yielded by a move m played in position p. Also, without loss of generality, we can safely assume that a

[1] Private communication with the authors of Gekisashi, YSS, and Shueso.

root position p is a position where the maximizing-player is to move. There exist many studies in the 1990s based on this idea of comparison [6].

To measure the total fitness over a set of training positions $\mathcal{P}$, we introduce the number of violations in all constraints:

$$J_{\text{step}}(\mathcal{P}, \theta) = \sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}(p)} T_{\text{step}} \left(\text{eval}(p.m, \theta) - \text{eval}(p.g_p, \theta) \right), \quad (1)$$

where $\mathcal{M}(p)$ is a set of legal moves in a position p and $T_{\text{step}}(x)$ is a step function of which the value is 1 in $x > 0$; and 0, otherwise. The minimum value of J_{step} is 0, when all constraints are satisfied. The maximum value of J_{step} is the number of constraints, $\sum_{p \in \mathcal{P}}(|\mathcal{M}(p)| - 1)$. Here, $|\mathcal{A}|$ denotes the cardinality of $\mathcal{A}$.

To utilize the gradient for minimization, we slightly modify the objective function:

$$J_{\text{oneply}}(\mathcal{P}, \theta) = \sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}(p)} T \left(\text{eval}(p.m, \theta) - \text{eval}(p.g_p, \theta) \right), \quad (2)$$

where $T(x)$ is a sigmoid function, $1/(1 + e^{-\alpha x})$. The slope of T is controlled by a constant $\alpha > 0$, and by increasing α, T becomes similar to T_{step}. The function T is differentiable w.r.t. x, while T_{step} is not. Thus, the function J_{oneply} is differentiable w.r.t. θ, provided the evaluation function $\text{eval}(p, \theta)$ is. A sigmoid function is confirmed to work well for training strong shogi programs [9]. Although previous studies did not discuss whether T should be a smooth approximation of the step function, there are other functions used instead of T. The function $(1/(1 + e^{R(x)}) - 1)^2$ with a heuristic rescaling function R was used in small chess experiments, as an error function for back propagation in neural networks [16]. A squared error loss function was used in preliminary experiments in Deep Thought, to fit search values to heuristic oracle values [14].

In practice, a one-ply search is not sufficiently powerful to be used for analyzing training positions. Thus, to learn accurate evaluation functions, a min-max search with a depth of more than 1 has been used in many of the previous studies on learning evaluation functions [14,2,16,9]. A new objective function incorporates a function $s(p, \theta)$, which is the min-max value identified by the min-max search for a position p, instead of $\text{eval}(p, \theta)$ in J_{oneply}:

$$J(\mathcal{P}, \theta) = \sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}(p)} T \left(s(p.m, \theta) - s(p.g_p, \theta) \right). \quad (3)$$

The differentiability of J now depends on the differentiability of s, as discussed in the next subsection. In our previous work, we made Eq. (3) our basic objective function and confirmed that it worked well in shogi [9]. In practice, it is also important to incorporate terms for constraints [9] in the objective function in order to avoid over-fitting or divergence. However, we will omit any discussion of these terms for simplicity, because they are independent of our purpose in this paper.

3.2 Partial Gradient of the Objective Function

To discuss the differentiability of $s(p, \theta)$, which is the min-max value of a position p identified by a min-max search, we shall analyze the min-max tree visited in the search. The root node of the min-max tree is p, and each node has a min-max value $v(n, \theta)$ defined as:

$$v(n,\theta) = \begin{cases} \mathrm{eval}(n,\theta) & \textit{if } n \textit{ is a leaf,} \\ \max_{c \in \mathcal{C}(n)} v(c,\theta) & \textit{if } n \textit{ is a max-node} \\ \min_{c \in \mathcal{C}(n)} v(c,\theta) & \textit{if } n \textit{ is a min-node.} \end{cases} \tag{4}$$

Here, $\mathcal{C}(n)$ is a set of direct successors of node n. By definition, $s(p, \theta) = v(p, \theta)$.

Let $\mathcal{L}(p)$ be a set of leaf nodes in the min-max tree of which the root is p. We focus on a subset of $\mathcal{L}(p)$, which plays an important role in computing $s(p, \theta)$:

$$\mathcal{L}'(p,\theta) = \{l \in \mathcal{L}(p) \mid (\mathrm{eval}(l,\theta){=}s(p,\theta)) \wedge (\forall n \in \mathcal{A}(l),\ v(n,\theta){=}s(p,\theta))\}\,, \tag{5}$$

where $\mathcal{A}(l)$ is the set of ancestor nodes of l in the tree. In other words, the path from $l \in \mathcal{L}'(p, \theta)$ to the root p is a *principal variation* (PV). This is because a child is considered to be the best choice in its parent node if the min-max value of the child is the same as that of the parent node. Note that $\mathcal{L}'$ contains at least one leaf node. We assume that the tree expanded by the search does not depend on θ. Thus, our discussion here is valid for variants of min-max searches with static termination conditions and pruning techniques based on depth, width, and/or a move itself. However, it is not valid for other techniques such as futility pruning. $\alpha\beta$ pruning is also turned off in order to find multiple PVs. A min-max tree may be a direct acyclic graph (DAG) with transpositions but it cannot be a cyclic graph.

Maximum Change in $v(n, \theta)$ w.r.t. θ: Assume that θ is slightly modified from θ_0 into θ_1, where the change is only in the i-th element s.t. $\theta_1(i) - \theta_0(i) = \delta$ and $\theta_1(j) - \theta_0(j) = 0$ for $j \neq i$. We first show that this change in θ cannot make a large change in the min-max value of each node if the change δ is sufficiently small.

Let $\mathcal{N}'(p, \theta)$ be the set of nodes that are on the path from the root p to each leaf on PV, $l \in \mathcal{L}'(p, \theta)$. From the definition of the min-max values in Eq. (4) and that of $\mathcal{L}'$ in Eq. (5), we have

$$\forall n \in \mathcal{N}'(p,\theta_0),\ v(n,\theta_0) = s(p,\theta_0), \tag{6}$$

$$\forall n \in \mathcal{N}'(p,\theta_0),\ \exists c \in \mathcal{C}(n),\ c \in \mathcal{N}'(p,\theta_0). \tag{7}$$

Then, there exists a constant $a > 0$ such that

$$\begin{cases} v(n,\theta_0) = s(p,\theta_0) > a + \max_{c \in (\mathcal{C}(n) \setminus \mathcal{N}'(p,\theta_0))} v(c,\theta_0) & (n\textit{:max-node}) \\ v(n,\theta_0) = s(p,\theta_0) < -a + \min_{c \in (\mathcal{C}(n) \setminus \mathcal{N}'(p,\theta_0))} v(c,\theta_0) & (n\textit{:min-node}) \end{cases} \tag{8}$$

for all internal nodes $n \in \mathcal{N}'(p, \theta_0)$ in the search tree. Here, $\mathcal{A} \setminus \mathcal{B}$ denotes the set difference that is $\{e | e \in \mathcal{A} \wedge e \notin \mathcal{B}\}$. The constant a represents the stableness of PV. In contrast, the maximum change in the min-max value of a node n is bound by the maximum change D_n in the leaves that are the descendants of n:

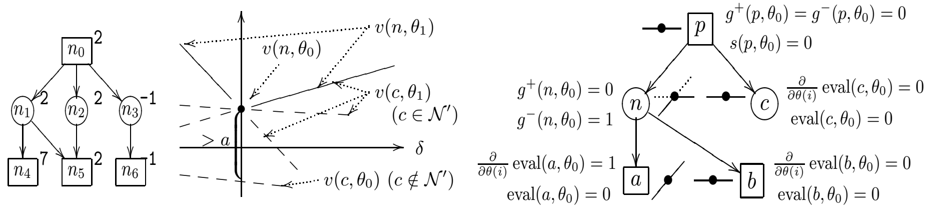


Fig. 1. Left: example of a min-max tree (graph) with a transposition at n_5, Center: $v(n,\theta_1)$ at a max-node $n \in \mathcal{N}'(p,\theta_0)$ depends only on the children $c \in \mathcal{N}'(p,\theta_0)$, Right: the partial derivative of $s(p,\theta)$ exists, but the value is not equal to that of a PV leaf $\frac{\partial}{\partial\theta(i)}\,\mathrm{eval}(a,\theta)$.

$$|v(n,\theta_1) - v(n,\theta_0)| \leq \max_{l \in \mathcal{L}(n)} |\,\mathrm{eval}(l,\theta_1) - \mathrm{eval}(l,\theta_0)| = D_n. \tag{9}$$

A formal proof of this bound by mathematical induction on a min-max tree, as well as a proof of the continuity of $v(n,\theta)$, will be published later [10]. For example, a small min-max tree in the left figure of Fig. 1 has two paths of principal variation; $n_0n_1n_5$, and $n_0n_2n_5$. In the figure, a max-node and min-node are denoted by a box and circle, respectively. Therefore, $\mathcal{L}'(n_0) = \{n_5\}$ and $\mathcal{N}' = \{n_0, n_1, n_2, n_5\}$.

Assume that each leaf value changes by at most 0.1. Note that such a δ always exists as our evaluation function is continuous. Then, it will be proven that $|v(n,\theta_1) - v(n,\theta_0)| \leq 0.1$ for each internal node n of height $1, 2, 3, ..$ in order: It is obvious for n_4, n_5, and n_6, then it can be proven for n_1, n_2 and n_3, and finally for n_0. Also, as is explained in the next paragraph, because $a = 2.9 < \min(|7-2|, |2+1|)$ satisfies Eq. (8) and $0.1 < \frac{a}{2}$, neither n_4 nor n_6 can become a new PV as a result of this change.

Partial Subgradient of $v(n,\theta)$: The new min-max value $v(n,\theta_1)$ on each node is identified as follows. Let the change δ be sufficiently small s.t. $D_p < a/2$. Under this condition, the new set of principal variations cannot be far different from the old set, and actually is a subset of the old set. Each internal node $n \in \mathcal{N}'(p,\theta_0)$ including the root p has the set of best children $(\mathcal{C}(n) \cap \mathcal{N}'(p,\theta_0)) \neq \emptyset$, from Eq. (7), and the new min-max value of each of such children is still better than that of any other children in the new tree with θ_1, from Eq. (8) and (9):

$$\begin{cases} \min\limits_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p,\theta_0))} v(c,\theta_1) > \max\limits_{c \in (\mathcal{C}(n) \setminus \mathcal{N}'(p,\theta_0))} v(c,\theta_1) & (n\textit{:max-node}) \\ \max\limits_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p,\theta_0))} v(c,\theta_1) < \min\limits_{c \in (\mathcal{C}(n) \setminus \mathcal{N}'(p,\theta_0))} v(c,\theta_1) & (n\textit{:min-node}) \end{cases} \tag{10}$$

for each internal node $n \in \mathcal{N}'(p,\theta_0)$. Thus, for $n \in \mathcal{N}'(p,\theta_0)$:

$$v(n,\theta_1) = \begin{cases} \mathrm{eval}(n,\theta_1) & (n\textit{:leaf}) \\ \max_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p,\theta_0))} v(c,\theta_1) & (n\textit{:max-node}) \\ \min_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p,\theta_0))} v(c,\theta_1) & (n\textit{:min-node}) \end{cases} \tag{11}$$

The center figure of Fig. 1 sketches an example of $v(n, \theta_1)$ changing with δ, where n is a max-node. There are the three best children with the value $v(n, \theta_0)$ when $\delta = 0$. Each value continuously (not always linearly) changes according to δ. While the best child changes depending on the sign of δ, it is always one of $c \in \mathcal{N}'$ when δ is sufficiently small. This is because the min-max values of other children $c \notin \mathcal{N}'$ are sufficiently less (by at least a) than $v(n, \theta_0)$ at $\delta = 0$.

Now, let us discuss the partial derivative of $v(n, \theta)$ where $n \in \mathcal{N}'(p, \theta_0)$, as a result of introducing two limits $g_i^+(n, \theta)$ and $g_i^-(n, \theta)$:

$$g_i^+(n, \theta_0) = \lim_{\delta \to +0} \frac{v(n, \theta_1) - v(n, \theta_0)}{\theta_1(i) - \theta_0(i)}, \;\; g_i^-(n, \theta_0) = \lim_{\delta \to -0} \frac{v(n, \theta_1) - v(n, \theta_0)}{\theta_1(i) - \theta_0(i)}. \tag{12}$$

It is obvious that $g_i^+(n, \theta) = g_i^-(n, \theta) = \frac{\partial}{\partial \theta(i)} \mathrm{eval}(n, \theta)$ for each leaf n. For an internal node n, by using Eq. (6) and Eq. (11), we obtain:

$$\begin{aligned} g_i^+(n, \theta) &= \begin{cases} \max_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p, \theta_0))} \; g_i^+(c, \theta) & (n\text{:}\mathit{max\text{-}node}) \\ \min_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p, \theta_0))} \; g_i^+(c, \theta) & (n\text{:}\mathit{min\text{-}node}) \end{cases} \\ g_i^-(n, \theta) &= \begin{cases} \min_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p, \theta_0))} \; g_i^-(c, \theta) & (n\text{:}\mathit{max\text{-}node}) \\ \max_{c \in (\mathcal{C}(n) \cap \mathcal{N}'(p, \theta_0))} \; g_i^-(c, \theta) & (n\text{:}\mathit{min\text{-}node}). \end{cases} \end{aligned} \tag{13}$$

For the root p, there always exist a leaf l_a and $l_b \in \mathcal{L}'(p)$ such that

$$g_i^+(p, \theta) = \tfrac{\partial}{\partial \theta(i)} \mathrm{eval}(l_a, \theta), \quad g_i^-(p, \theta) = \tfrac{\partial}{\partial \theta(i)} \mathrm{eval}(l_b, \theta). \tag{14}$$

Thus, the partial derivative of $s(p, \theta)$ exists if $g_i^+(p, \theta) = g_i^-(p, \theta)$:

$$\frac{\partial}{\partial \theta(i)} s(p, \theta) = \frac{\partial}{\partial \theta(i)} \mathrm{eval}(l_a, \theta). \tag{15}$$

It is obvious that the partial derivative, Eq. (15), exists if $|\mathcal{L}'(p)| = 1$ or if all the partial derivatives of the leaves in $\mathcal{L}'$ are identical. Also, even if the leaves have different partial derivatives, the partial derivative of s may still exist, as sketched in the right figure of Fig. 1, where the partial derivative is 1 for a and 0 for b and c.

Finally, the partial derivative of the objective function J is defined if all the positions are partially differentiable:

$$\frac{\partial J(\mathcal{P}, \theta)}{\partial \theta(i)} = \sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}(p)} \left. \frac{\mathrm{d}\, T(x)}{\mathrm{d}\, x} \right|_{x = s(p.m, \theta) - s(p.g_p, \theta)} \frac{\partial \left(s(p.m, \theta) - s(p.g_p, \theta) \right)}{\partial \theta(i)}. \tag{16}$$

A previous study implicitly assumes $|\mathcal{L}'| = 1$ [16] or uses this partial derivative as an approximation for general cases without further analysis [9].

Practical Issues: Practical game-tree search methods are optimized to find *one of* the principal variations and are not suitable for finding all the leaves of PVs. So, our interest is on the magnitude of errors accumulated when we stick to

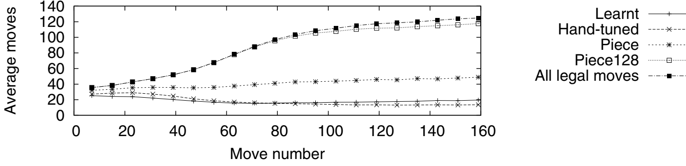


Fig. 2. Number of legal moves and sibling moves with similar evaluation

using the leaf of the PV found by the search, and interpreting it as l_a in Eq. (15). The errors will be negligible when $|\mathcal{L}'(p,\theta)| = 1$ for most positions p. However, as shown in the experiments in the next section, $|\mathcal{L}'(p,\theta)|$ is often more than 1 especially at the beginning of learning. A different optimistic scenario is that for most positions p and dimension i, $\frac{\partial}{\partial\theta(i)}\,\mathrm{eval}(l,\theta)$ is equal or similar for each $l \in \mathcal{L}'(p,\theta)$. Also, the partial derivative of $\frac{\partial}{\partial\theta(i)}\,(s(p.m,\theta) - s(p.g_p,\theta))$ may exist even if the partial derivative of $s(p.g_p,\theta)$ does not. However, such cases are rare in practice.

A second important issue is the smallness of δ. We have so far assumed that the parameter vector θ consists of real numbers. However, the parameters in a practical evaluation function are of the integral type so that the game-tree search will be efficient. Obviously, the smallest δ is 1 in integral type. Eq. (10) can no longer be safely assumed when 1 is relatively larger than a. In such cases, a new search value with θ_1 cannot be produced by any of the previous PVs with θ_0, i.e., $\forall l \in \mathcal{L}'(p,\theta_0),\ s(p,\theta_1) \neq \mathrm{eval}(l,\theta_1)$. Consequently, even when we have a precise gradient, the objective function J may become worse after updating with integer parameters.

4 Experimental Results

This section shows experimental results on the accuracies of the approximation of the gradients using the leaf position. It also discusses their effectiveness at controlling the search results when the parameters are represented by integer.

4.1 Game of Shogi and Shogi Programs

The experiments were conducted on a computer program for playing shogi, a Japanese variant of chess [11]. The rules of shogi are similar to those of chess in that the $\alpha\beta$ search guided by a heuristic evaluation function works well. A unique regulation in shogi is the "dropping" rule whereby captured pieces can be placed again on almost any empty square on the board by the capturing player. Practical endgame databases are thus unavailable in shogi, and the branching factor and the average move numbers for a game are greater than those of chess. We chose to experiment with GPSShogi [12], which is the winner of the CSA Championship in 2009. CSA Championship is the most authoritative tournament in computer shogi. The source codes are publicly available on the Web [15].

Four different evaluation functions were used: (a) *Learnt* is the main evaluation function in GPSShogi revision 2590. It has about 8 million parameters, and the number of non-zero values after learning is about 1.4 million. The values should be near optimal since they were adjusted by minimizing Eq. (3) with a difference in $T(x)$; the hinge loss was used instead of the sigmoid loss. (b) *Hand-tuned* is an old evaluation function that had been used until 2008. The features were different from those of *Learnt*, and the parameters were hand-tuned, except for a few opening features. The parameters are considered to be reasonable but far from the optimal because the winning ratio in self-play is almost 70% if the majority of the parameters were adjusted by learning. (c) *Piece* had the same features as *Learnt* but the parameters were set to zero, except for 14 piece values. This was done to enable analysis of the gradients at the beginning of learning. (d) *Piece128* was similar to *Piece* but the parameters for all piece values were set to 128. This was done to enable analysis of an extreme start-point of learning.

The training set consisted of 47,538 scores of professional games. The search in $s(n, \theta)$ was a full width search of depth 1 (that is, a total 2-ply search if counted from a position in the game records) with a quiescence expansion. While a 1- to 3-ply search was used in similar experiments in chess [1,14,16], a 1-ply search was used here and in the training of a strong program [9] due to the large branching factors of shogi. For efficiency, an $\alpha\beta$ window $s(g_p, \theta) \pm w$ was used in the search for moves other than g_p if appropriate. The actual value w was different for each evaluation function and set to twice the pawn value. Fig. 2 shows how the average number of legal moves increases with the move number, as well as the average number of moves inside the window. Moves outside the window were ignored in computing J, as the gradient of $T(x)$ is almost 0 if $|x|$ is sufficiently large. The constant α in $T(x)$ was set to be $0.0273 \cdot 256/w$, where 0.0273 is the learning parameter used in Bonanza [8]. The operating system was Linux amd64 (Debian Squeeze). Several computers were used but details are not relevant because the cpu time is not discussed here. It took eight X5570 cores a few hours to identify all search values needed for J for each parameter set.

4.2 Existence of Partial Derivative

Table 1 shows various estimates of the upper bound of the frequency of positions where the partial derivative of $s(p, \theta)$ does not exist, for each evaluation function.

First, the frequency for which a position has multiple principal variations was measured for all primary positions, $p.g_p$, in the training data. We did this because Eq. (16) can be safely used if $|\mathcal{L}'| = 1$ for all positions. The second column, "PV path", in Table 1, shows the frequency for which a position has multiple paths in PV. The third column, $|\mathcal{L}'|$, shows the frequency for which a position has multiple leaves in PV. Both frequencies are similar but may differ in the case of a transposition, as sketched in the left tree in Fig. 1. It is natural that the estimation by $|\mathcal{L}'|$ has a lower frequency. Both frequencies are quite low for *Learnt*. Thus, we can expect that errors introduced by such positions will be negligible for this set of features and parameters on the condition that the distribution of the values of the gradients of eval is similar in all training positions.

Table 1. Various estimates of the upper bound of the frequency for which the min-max value is not differentiable

	PV path(%)	$\lvert\mathcal{L}'\rvert$ (%)	Pawn (%)	(non zero)	Plance (%)	(non zero)
Learnt	2.72	0.486	0.0414	0.0737	0.0004	0.0102
Hand-tuned	20.1	14.4	4.09	7.15	0.338	8.07
Piece	88.2	87.5	0.149	0.267	0.0285	0.697
Piece128	99.3	99.3	71.3	87.2	7.77	78.4

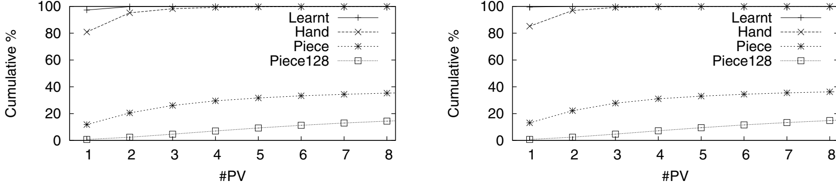


Fig. 3. Cumulative frequency of the number of PVs (left: by paths, right: by leaves)

However, these frequencies become non-negligible for *Hand-tuned* and quite high for *Piece* and *Piece128*. The cumulative frequencies of $|\mathcal{L}'|$ are shown in Fig. 3. The horizontal axis represents $|\mathcal{L}'|$, and the vertical axis is for the frequency of positions of which $|\mathcal{L}'| \leq x$. One can see that the cumulative frequency is still less than 40% and 20% in *Piece* and *Piece128*, respectively, even for $|\mathcal{L}'| \leq 8$.

Even when $|\mathcal{L}'| > 1$, the min-max value of p is still partially differentiable w.r.t. such a $\theta(i)$ that the partial gradient of $\frac{\partial}{\partial\theta(i)}\,\mathrm{eval}(n, \theta)$ has the same value for all $n \in \mathcal{L}'(p, \theta)$. To estimate the frequency of such cases, the partial gradient of two major features, the pawn value and promoted-lance (plance) value, were analyzed. The fourth to seventh column in Table 1 show the frequency for which different partial gradients exist for leaves in $\mathcal{L}'$. The frequencies were separately measured for all positions and for positions where any leaf in $\mathcal{L}'$ had non-zero partial gradients. The latter statistics are less dependent than the former ones are on how frequently the feature occurs in the training positions. The cumulative values of both measurements are shown in Fig. 4. Again, for all statistics, the frequency of non-differentiable positions is rather low for *Learnt*. Also, the frequencies for *Hand-tuned* and *Piece* are quite low here. In contrast, the frequencies are still over 87% and 78% for *Piece128*. This indicates that reasonable piece values, rather than a single value for all pieces, are preferable at the beginning of learning.

4.3 Effectiveness of Gradient Descent

The next question is whether the objective function J in Eq. (3) will decrease after an update of θ by an approximation of the gradient of Eq. (16).

First, all derivatives in Eq. (16) for all features in each evaluation function were computed and stored. Fig. 5 shows the distribution of the absolute values

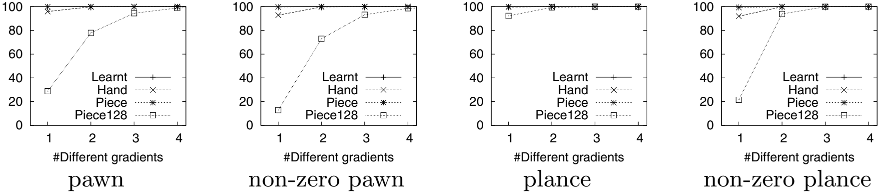


Fig. 4. Cumulative frequency of the number of different gradients in $\mathcal{L}'$ (feature: pawn and plance, positions: all and non-zero gradient)

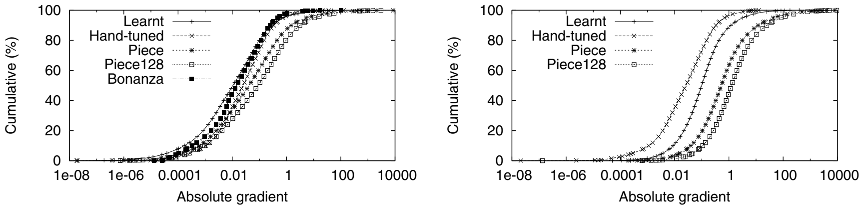


Fig. 5. Distribution of absolute value of gradients (left: all features, right: non-zero features)

of the derivatives. The vertical axis is for the absolute value, and the horizontal axis is for the cumulative percentage of that value when the absolute values are in nondecreasing order. For a reasonable comparison of absolute values, derivatives for *Hand-tuned* were divided by 16 and those of the other three evaluation functions were divided by 341, so that the pawn value would be scaled to 128 in the four evaluation functions. The left figure is for non-zero derivatives, and the right one is for derivatives of which the corresponding parameter is not zero. The results of the latter tests for *Learnt* directly applied for *Piece* and *Piece128*, as they used the same set of features as *Learnt*. The right figure also shows data for the evaluation function in BONANZA [8] with the same training data, 256 as the search window w and 0.0273 as α in T.

The range of the absolute values is about 10^{-8} to 10^{5}. Apparently, the derivatives for the non-zero features in the right figure have a larger variance than those for all features. This is because features of which the parameter is zero after learning or hand-tuning tend to have a derivative near zero. Derivatives of *Learnt* were significantly smaller than those of *Piece* or *Piece128*. Since these three evaluation functions have the same feature set, this difference suggests sufficient learning decreased the absolute values of the derivatives.

An important observation is that the standard update method proportional to the gradients, such as $\theta^{t+1} = \theta^{t} - \alpha\nabla\theta^{t}$, is not suitable in this situation. The largest derivative is more than 10^{7} times the smallest derivative even when the values of both extreme end were omitted. As long as the parameters in an evaluation function are represented by integers, this means that the largest parameter will be updated by 10^{7} if one updates the smallest parameter by 1, which is the minimum δ of the integral type. GPSSHOGI uses hand-crafted

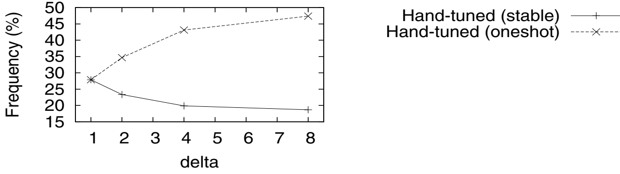


Fig. 6. Frequency of features for which J decreased for various δ: {1, 2, 4, 8}

features based on knowledge of shogi [12]; these features are totally different from the brute-force combinations of piece-features in Bonanza. Thus, this phenomenon can be expected to happen with various evaluation functions.

For 1,024 randomly selected features, the changes in the objective function J were measured for $\delta = 1, 2, 4, 8$. Only *Hand-tuned* was used in this experiment, because our computational resources were limited. The horizontal axis of Fig. 6 is for the δ. The vertical axis is for the frequency of which the number of features decreased J after one update ("one-shot") or for all updates within δ ("stable"). This time, $J(\mathcal{P}, \theta_1)$ was computed with θ_1 assuming that $\mathcal{L}'(, \theta_1)$ are the same as $\mathcal{L}'(, \theta_0)$ for all positions. By increasing δ, the frequency of "stable" decreases while the frequency of "one-shot" increases. This means that the slope of J around the parameter of *Hand-tuned* is often non-monotonic.

Finally, we analyzed the change in J with the exact $\mathcal{L}'(, \theta_1)$. 164 stable features were randomly selected among ones decreased J with any $\delta = 1, 2, 4, 8$ in the previous experiment. Again, due to limited computational resources, J before and after the update were measured with an 8,000 record subset of the training data. The frequency that J decreased was 28.7% and 21.3% for δ with 1 and 8, respectively. The frequency that J increased was 28.7% and 45.1% for δ with 1 and 8, respectively. Since the derivatives were computed by using the full training records, it is not surprising that the frequency did not reach 100%. However, the differences between 28.7% and 21.3% and between 28.7% and 45.1% indicate that $\delta = 1$ is preferable to $\delta = 8$ for this update.

5 Concluding Remarks

This paper discussed the gradients of the search results as to how the min-max value changes along with a change in a parameter vector θ in an evaluation function. By analyzing the variations of min-max values in a search tree, it was shown that the variation of the min-max value is, in general, determined by the gradients of the two special leaves as in Eq. (14) depending on a search tree. Recent supervised and reinforcement learning methods often use the partial derivative of the leaf node in a PV found by a tree search [14,16,9,2]. However, it was found that this partial derivative does not always equal the partial derivative of the min-max value at the root even for positions where the root value is partially differentiable. Thus, careful utilization of the subgradients analyzed in this paper may lead to improvements of these methods.

The experiments on a shogi program with about a million of parameters showed how often the search value is differentiable as well as the accuracy of the approximation of the gradients using the leaf position. In most cases, the gradients of the leaf of PV seemed to be good guides for the subgradients of the root value. However, large errors occurred when the evaluation function valued all pieces equally. Moreover, as long as parameters are represented by integers, 1 or a similar small value would be a suitable step-size in the updates of the parameters [9] rather than more popular steps proportional to their gradients. Here, we presented two new empirical supports for the effectiveness of small step-sizes: (1) the wide range of the distribution of the absolute values of derivatives and (2) the statistics in changes in the objective function after updating.

References

1. Anantharaman, T.: Evaluation tuning for computer chess: Linear discriminant methods. ICCA Journal 20, 224–242 (1997)
2. Baxter, J., Tridgell, A., Weaver, L.: Learning to play chess using temporal-differences. Machine Learning 40, 242–263 (2000)
3. Beal, D.F., Smith, M.C.: Temporal difference learning applied to game playing and the results of application to shogi. Theoretical Computer Science 252, 105–119 (2001)
4. Buro, M.: Improving heuristic mini-max search by supervised learning. Artificial Intelligence 134, 85–99 (2002)
5. Campbell, M., Hoane Jr., A.J., Hsu, F.H.: Deep blue. Artificial Intelligence 134, 57–83 (2002)
6. Fawcett, T.E.: Feature Discovery for Problem Solving Systems. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst (1993)
7. Fürnkranz, J.: Machine learning in games: a survey. In: Machines that Learn to Play Games, pp. 11–59. Nova Science Publishers, Commack (2001)
8. Hoki, K.: (2005) (in Japanese), http://www.geocities.jp/bonanza_shogi/
9. Hoki, K.: Optimal control of minimax search results to learn positional evaluation. In: GPW 2006, pp. 78–83 (2006) (in Japanese)
10. Hoki, K., Kaneko, T.: Large-scale optimization of evaluation functions with minimax search (in preparation)
11. Iida, H., Sakuta, M., Rollason, J.: Computer shogi. Artificial Intelligence 134, 121–144 (2002)
12. Kaneko, T.: Recent improvements on computer shogi and GPS-Shogi. Journal of Information Processing Society of Japan 50, 878–886 (2009) (in Japanese)
13. Marsland, T.: Evaluation function factors. ICCA Journal 8, 47–57 (1985)
14. Nowatzyk, A.: (2000), http://tim-mann.org/DT_eval_tune.txt
15. Tanaka, T., Kaneko, T.: (2003), http://gps.tanaka.ecc.u-tokyo.ac.jp/gpsshogi/
16. Tesauro, G.: Comparison training of chess evaluation functions. In: Machines that Learn to Play Games, pp. 117–130. Nova Science Publishers (2001)
17. Tesauro, G.: Programming backgammon using self-teaching neural nets. Artificial Intelligence 134, 181–199 (2002)
18. Veness, J., Silver, D., Uther, W., Blair, A.: Bootstrapping from game tree search. In: Advances in Neural Information Processing Systems 22, pp. 1937–1945 (2009)

Approximating Optimal Dudo Play with Fixed-Strategy Iteration Counterfactual Regret Minimization

Todd W. Neller and Steven Hnath

Gettysburg College, Dept. of Computer Science, Gettysburg, Pennsylvania, 17325, USA
tneller@gettysburg.edu, steve.hnath@gmail.com
http://cs.gettysburg.edu/~tneller

Abstract. Using the bluffing dice game Dudo as a challenge domain, we abstract information sets by an imperfect recall of actions. Even with such abstraction, the standard Counterfactual Regret Minimization (CFR) algorithm proves impractical for Dudo, since the number of recursive visits to the same abstracted information sets increase exponentially with the depth of the game graph. By holding strategies fixed across each training iteration, we show how CFR training iterations may be transformed from an exponential-time recursive algorithm into a polynomial-time dynamic-programming algorithm, making computation of an approximate Nash equilibrium for the full 2-player game of Dudo possible for the first time.

1 Introduction

In recent years, Counterfactual Regret Minimization (CFR) has proven to be an important innovation in advancing optimal strategy approximation for large extensive game-trees of partially-observable stochastic games (POSGs) such as Texas Hold'em Poker [10,5,7]. Imperfect recall of actions is one of the means of abstracting such large extensive games. The approximation of optimal play with this abstraction has led to play performance improvements with the non-abstracted game [9]. In this paper, we demonstrate how a straightforward modification to CFR, using imperfect recall of actions, reduces individual training iterations from exponential to polynomial time complexity, allowing the first computation of an approximately optimal strategy for the full game of 2-player Dudo. After introducing Dudo, we will (1) review the imperfect recall of actions and CFR, (2) motivate and introduce Fixed-Strategy Iteration CFR (FSICFR), and (3) compare the performance of CFR and FSICFR. Finally, we will (4) discuss four open questions.

1.1 The Game of Dudo

Dudo is a bluffing dice game thought to originate from the Inca Empire circa 15th century. Many variations exist in both folk and commercial forms. The rule set we use from [3] is perhaps the simplest representative form, and is thus most easily accessible to both players and researchers. Liar's Dice, Bluff, Call My Bluff, Perudo, Cacho, Cachito are names of variations[1].

[1] In some cases, e.g., Liar's Dice and Cacho, there are different games of the same name.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 170–183, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Dudo has been a popular game through the centuries. From the Inca Empire, Dudo spread to a number of Latin American countries, and is thought to have come to Europe via Spanish conquistadors [6]. It is said to have been "big in London in the 18th century" [2]. Richard Borg's commercial variant, published under the names Call My Bluff, Bluff, and Liar's Dice, won the prestigious Spiel des Jahres (German Game of the Year) in 1993. On BoardGameGeek.com[2], the largest website for board game enthusiasts, Liar's Dice is ranked 270/53298 (i.e., top 0.5%)[3]. Although a single, standard form of the game has not emerged, there is strong evidence of the persistence of the core game mechanics of this favorite bluffing dice game since its creation.

Rules: Each player is seated around a table and begins with five standard six-sided dice and a dice cup. Dice are lost with the play of each round, and the object of the game is to be the last player remaining with dice. At the beginning of each round, all players simultaneously roll their dice once, and carefully view their rolled dice while keeping them concealed from other players. The starting player makes a claim about what the players have *collectively* rolled, and all players clockwise and in turn continue by either making a stronger claim or challenging the previous claim, by declaring "Dudo" (Spanish for "I doubt it."). A challenge ends the round, and then players lift their cups, and one of the two players involved in the challenge loses dice. Lost dice are placed in full view of players.

Claims consist of a positive number of dice and a rank of those dice, e.g., two 5's, seven 3's, or two 1's. In Dudo, the rank of 1 is *wild*, meaning that dice rolls of rank 1 are counted in totals for other ranks as well. We will denote a claim of n dice of rank r as $n \times r$. In general, one claim is stronger than another claim if there is an increase in rank and/or number of dice. That is, a claim of 2×4 may, for example, be followed by 2×6 (increase in rank) or 4×3 (increase in number). The exception to this general rule concerns claims of wild rank 1. Since 1's count for other ranks and other ranks do not count for 1's, 1's as a rank occur with half frequency in counts and are thus considered doubly strong in claims. So, in the claim ordering, 1×1, 2×1, and 3×1 immediately precede 2×2, 4×2, and 6×2, respectively.

Mathematically, one may enumerate the claims in order of strength by defining $s(n, r)$, the strength of claim $n \times r$, as follows:

$$s(n,r) = \begin{cases} 5n - \lfloor \frac{n}{2} \rfloor - r - 7 & \text{if } r \neq 1 \\ 11n - 6 & \text{if } r = 1 \text{ and } r \leq \lfloor \frac{d_{\text{total}}}{2} \rfloor \\ 5d_{\text{total}} + n - 1 & \text{if } r = 1 \text{ and } r > \lfloor \frac{d_{\text{total}}}{2} \rfloor \end{cases} \quad (1)$$

where d_{total} is the total number of dice in play. Thus for 2 players with 1 die each, the claims would be numbered as follows.

Strength $s(n,r)$	0	1	2	3	4	5	6	7	8	9	10	11
Claim $n \times r$	1×2	1×3	1×4	1×5	1×6	1×1	2×2	2×3	2×4	2×5	2×6	2×1

Play proceeds clockwise from the round-starting player with claims of strictly increasing strength until one player challenges the previous claimant with "Dudo". At this

[2] http://www.boardgamegeek.com

[3] as of August 17th, 2011.

point, all cups are lifted, dice of the claimed rank (including wilds) are counted and compared against the claim. For example, assume that Ann, Bob, and Cal are playing Dudo, and Cal challenges Bob's claim of 7×6. There are three possible outcomes.

- **The actual rank count exceeds the challenged claim.** In this case, the challenger loses a number of dice equal to the difference between the actual rank count and the claim count. Example: Counting 6's and 1's, the actual count is 10. Thus, as an incorrect challenger, Cal loses $10 - 7 = 3$ dice.
- **The actual rank count is less than the challenged claim.** In this case, the challenged player loses a number of dice equal to the difference between the claim count and the actual rank count. Example: Counting 6's and 1's, the actual count is 5. Thus, as a correctly challenged claimant, Bob loses $7 - 5 = 2$ dice.
- **The actual rank count is equal to the challenged claim.** In this case, every player except the challenged player loses a single die. Example: Counting 6's and 1's, the actual count is indeed 7 as Bob claimed. In this special case, Ann and Cal lose 1 die each to reward Bob's exact claim.

In the first round, an arbitrary player makes the first claim. The winner of a challenge makes the first claim of the subsequent round. When a player loses all remaining dice, the player loses and exits the game. The last remaining player is the winner. The following table provides a transcript of an example 2-player game with "1:" and "2:" indicating information relevant to each player.

Round	Actions	Revealed Rolls	Result
1	1:"2×6", 2:"3×6", 1:"5×6", 2:"Dudo"	1:12566 2:23556	1:loses 1 die
2	2:"3×6", 1:"4×5", 2:"5×5", 1:"Dudo"	1:3555 2:23455	1:loses 1 die
3	2:"3×6", 1:"Dudo"	1:356 2:24466	1:loses 1 die
4	2:"2×2", 1:"3×2", 2:"Dudo"	1:12 2:13456	2:loses 1 die
5	1:"2×6", 2:"3×2", 2:"Dudo"	1:26 2:1222	1:loses 2 dice

2 Imperfect Recall and Counterfactual Regret Minimization

In this work, we restrict our attention to two-player Dudo, yet even with this simplification, we will show that the number of information sets poses difficulties for modern computing.

Since Dudo is divided into rounds, the play environment is episodic in nature and information from previous rounds is not relevant for the decision at hand[4]. An *information set* consists of sequences of moves and chance outcomes consistent with a player's state of knowledge. Thus, a Dudo information set consists of (1) a history of claims from the current round, (2) the player's private roll information, and (3) the number of opponent dice. Let d_1, d_2, and $d_{\max}$ denote the number of current player 1 dice, the number of opponent player 2 dice, and the maximum number of dice per player, respectively. In general, the number of information sets for d_1 and d_2 in a two-player game is then the

[4] The focus here is on optimal play. When seeking rather to model and exploit a suboptimal opponent, play information from previous rounds would, of course, be relevant.

product of the number of possible claim histories $2^{6(d_1+d_2)}$ (i.e., the size of the power set of possible claims), and the number of possible rolls $\binom{d_1+5}{d_1}$.

However, in Dudo it is impossible to have $d_1 < d_2 = d_{\max}$ with an even number of claims. An even number of claims in a claim history implies that the current player started the current round, and thus either the current round is the first round, or the current player won the previous round-ending challenge. However, $d_1 < d_{\max}$ implies this is not the first round, and $d_2 = d_{\max}$ implies the opponent could not have lost a previous challenge. Therefore, the current player lost the previous challenge, leading us to a contradiction. Since exactly half of power sets have an even-numbered size, we must reduce such information set counts by half. A symmetric argument applies for $d_2 < d_1 = d_{\max}$ with an odd number of claims. Our computation counting the number of power sets then becomes

$$\sum_{d_1,d_2\in[1,5]} \begin{cases} 2^{6(d_1+d_2)-1}\binom{d_1+5}{d_1} & \text{if } d_1 < d_2 = d_{\max} \text{ or } d_2 < d_1 = d_{\max}, \\ 2^{6(d_1+d_2)}\binom{d_1+5}{d_1} & \text{otherwise.} \end{cases}$$
$$= 294,021,177,291,188,232,192.$$

Thus, a full 2-player, 5-versus-5 dice game of Dudo has over 294 quintillion information sets.

2.1 Imperfect Recall of Actions

For common modern machines, 2.9×10^{20} information sets is too large to iterate over for convergence of mixed strategies. As with successful computational approaches to Texas Hold'em Poker, we abstract information sets in order to reduce the problem size. We then solve the abstraction and apply the abstracted policy to the original game. For Dudo, the growth rate of possible claim sequences is most responsible for the overall growth of the extensive game tree. We also note that the later claims of a round are less easily supported and thus more often contain reliable information.

Since more recent claims tend to be more important to the decision at hand, our chosen means of abstraction is to form abstract information sets that recall up to m previous claims. For example, consider this 5-vs.-5 round claim sequence: 1×5, 2×5, 4×2, 5×4, 3×1, 6×4, 7×2. For a claim memory limit of $m = 3$, the 5-vs.-5 round information set for each of these decisions would be enumerated according to the current player dice roll enumeration and the enumeration of the up-to-3 most recent claims: $\{\}, \{1 \times 5\}, \{1 \times 5; 2 \times 5\}, \{1 \times 5; 2 \times 5; 4 \times 2\}, \ldots, \{3 \times 1; 6 \times 4; 7 \times 2\}$.

An imperfect recall of actions allows us to trade off fine distinction of judgment for computational space and time requirements. Figure 1 shows how the number of abstract information sets varies according to the memory limit m and the number of dice for each player. As we will see, we can apply imperfect recall of actions without significantly impacting play performance.

2.2 Counterfactual Regret Minimization

Counterfactual Regret Minimization (Algorithm 1), while defined for general extensive game trees, can also be applied to abstracted information sets. We will now summarize

m	**Information Sets**
1	57626
2	2069336
3	21828536
4	380033636
5	2751474854
all	2.9×10^{20}

(a) No. of abstracted info. sets with varying memory limit m.

	Opponent Dice				
Dice	**1**	**2**	**3**	**4**	**5**
1	1794	5928	13950	27156	43056
2	20748	48825	95046	163947	241962
3	130200	253456	437192	693504	971264
4	570276	983682	1560384	2327598	3132108
5	159012	217224	284508	360864	9084852

(b) Information sets for each round with memory limit $m = 3$.

Fig. 1. Number of abstracted information sets varying recall limit and number of dice in round

the Counterfactual Regret Minimization (CFR) algorithm, directing the reader to [10] and [5] for detailed descriptions and proofs. At each player node recursively visited in a training iteration, a mixed strategy is computed according to the regret-matching equation, for which we now provide notation and which we define in a manner similar to [5].

Let A denote the set of all game *actions*. Let I denote an *information set*, and $A(I)$ denote the set of legal actions for information set I. Let t and T denote time steps. (Within both algorithms, t is with respect to each information set and is incremented with each visit to the information set.) A *strategy* σ_i^t for player i maps each player i information set I_i and legal player i action $a \in A(I_i)$ to the probability that the player will choose a in I_i at time t. All player strategies together at time t form a *strategy profile* σ^t. We refer to a strategy profile that excludes player i's strategy as σ_{-i}. Let $\sigma_{I \to a}$ denote a strategy equivalent to σ, except that action a is always chosen in information set I.

Let $\pi^\sigma(h)$ be the reach probability of game history h with strategy profile σ. Further, let $\pi^\sigma(I)$ be the reach probability of reaching information set I through all possible game histories in I, i.e., $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$. The counterfactual reach probability of information state I, $\pi^\sigma_{-i}(I)$, is the probability of reaching I with strategy profile σ except that, counter to the fact of σ, we treat current player i actions to reach the state as having probability 1. In all situations we refer to as "counterfactual", one treats the computation as if player i's strategy was modified to have intentionally played to information set I_i. Put it otherwise, we *exclude* the probabilities that factually came into player i's play from the computation.

Let Z denote the set of all terminal game histories. Then proper prefix $h \sqsubset z$ for $z \in Z$ is a nonterminal game history. Let $u_i(z)$ denote the utility to player i of terminal history z. Define the *counterfactual value* at nonterminal history h as:

$$v_i(\sigma, h) = \sum_{z \in Z, h \sqsubset z} \pi^\sigma_{-i}(h) \pi^\sigma(h, z) u_i(z). \tag{2}$$

The *counterfactual regret* of not taking a at history h is defined as:

$$r(h, a) = v_i(\sigma_{I \to a}, h) - v_i(\sigma, h). \tag{3}$$

The *counterfactual regret* of not taking action a at information set I is then:

$$r(I, a) = \sum_{h \in I} r(h, a). \tag{4}$$

The difference between the value of always choosing action a and the average value of the node strategy is an action's regret, which is then weighted by the probability that other player(s) will play to reach the node. This is then averaged over all time steps. If we define the nonnegative counterfactual regret $r_i^{T,+}(I, a) = \max(r_i^T(I, a), 0)$, then Hart and Mas-Colell's regret-matching equation [1] for the strategy at time $T + 1$ is:

$$\sigma_i^{T+1}(I, a) = \begin{cases} \frac{r_i^{T,+}(I,a)}{\sum_{a \in A(I)} r_i^{T,+}(I,a)} & \text{if } \sum_{a \in A(I)} r_i^{T,+}(I, a) > 0 \\ \frac{1}{|A(I)|} & \text{otherwise.} \end{cases} \tag{5}$$

For each player node, this equation is used to compute action probabilities in proportion to the positive average regrets. For each action, CFR then computes associated child nodes for each player, and computes utilities of such actions through recursive calls with updated probabilities for reaching such nodes through the current node. Regrets are computed from the returned values, and the value of playing to the current node is finally computed and returned.

The CFR algorithm is presented in detail in Algorithm 1. The parameters to CFR are the history of actions, the learning player, the time step, and the reach probabilities for players 1 and 2, respectively. Variables beginning with v are for local computation and are not computed according to the previous equations for counterfactual value. In lines 13, 15, and 20, $P(h)$ is the active player after history h. In lines 14 and 16, ha denotes history h with appended action a. In line 22, π_{-i} refers to the counterfactual reach probability of the node, which in the case of players $\{1, 2\}$ is the same as reach probability π_{3-i}. In line 32, $\emptyset$ refers to the empty history.

The average strategy profile at information set I, $\bar{\sigma}^T$, approaches an equilibrium as $T \to \infty$. The average strategy at information set I, $\bar{\sigma}^T(I)$, is obtained by normalizing s_I over all actions $a \in A(I)$. What is most often misunderstood about CFR is that this *average* strategy profile, and not the final strategy profile, is what converges to a Nash equilibrium.

3 Fixed-Strategy Iteration CFR

The technique we introduce in this paper, Fixed-Strategy Iteration Counterfactual Regret Minimization (FSICFR), is a significant structural modification to chance-sampled CFR yet convergence for both relies on the regret-matching equation (5) which is common to both algorithms.

Essentially, CFR traverses extensive game subtrees, recursing forward with reach probabilities that each player will play to each node (i.e., information set) while maintaining history, and backpropagating values and utilities used to update parent node action regrets and thus future strategy.

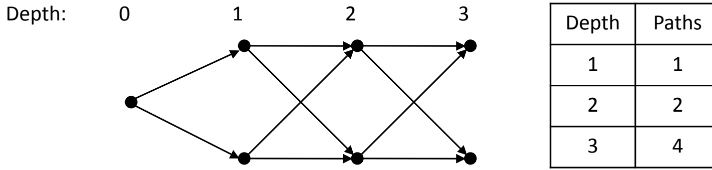
When applying chance-sampled CFR to abstracted Dudo, we observed that the number of recursive visits to abstracted player nodes grew exponentially with the depth

Algorithm 1. Counterfactual Regret Minimization

```
1: Initialize cumulative regret tables: ∀I, r_I[a] ← 0.
2: Initialize cumulative strategy tables: ∀I, s_I[a] ← 0.
3: Initialize initial profile: σ^1(I, a) ← 1/|A(I)|
4:
5: function CFR(h, i, t, π_1, π_2):
6: if h is terminal then
7:     return u_i(h)
8: end if
9: Let I be the information set containing h.
10: v_σ ← 0
11: v_{σ_{I→a}}[a] ← 0 for all a ∈ A(I)
12: for a ∈ A(I) do
13:     if P(h) = 1 then
14:         v_{σ_{I→a}}[a] ← CFR(ha, i, t, σ^t(I, a) · π_1, π_2)
15:     else if P(h) = 2 then
16:         v_{σ_{I→a}}[a] ← CFR(ha, i, t, π_1, σ^t(I, a) · π_2)
17:     end if
18:     v_σ ← v_σ + σ^t(I, a) · v_{σ_{I→a}}[a]
19: end for
20: if P(h) = i then
21:     for a ∈ A(I) do
22:         r_I[a] ← r_I[a] + π_{−i} · (v_{σ_{I→a}}[a] − v_σ)
23:         s_I[a] ← s_I[a] + π_i · σ^t(I, a)
24:     end for
25:     σ^{t+1}(I) ← regret-matching values computed using Equation 5 and regret table r_I
26: end if
27: return v_σ
28:
29: function Solve():
30: for t = {1, 2, 3, . . . , T} do
31:     for i ∈ {1, 2} do
32:         CFR(∅, i, t, 1, 1)
33:     end for
34: end for
```

of the tree. Consider Figure 2, a straightforward directed acyclic graph (DAG). With increasing depth, the number of possible paths to a node grows exponentially. If we continue the DAG pattern to greater depth, we have a linear growth of nodes and exponential growth of paths.

Similarly, in the game of Dudo with imperfect recall of actions, there are many paths to an abstracted information set. (For the remainder of the paper, assume that information sets are abstracted information sets, and that player nodes represent such abstracted information sets.) Each added die linearly grows the number of possible claims, which linearly grows the depth of the game-DAG and exponentially grows the number of paths to each information set. As will be seen in the experimental results, this exponential growth makes the application of CFR to the full game of Dudo impractical.



Depth	Paths
1	1
2	2
3	4

Fig. 2. Example directed acyclic graph

Fixed-Strategy Iteration CFR (FSICFR) divides the recursive CFR algorithm into two iterative passes, one forward and one backward, through a DAG of nodes. On the forward pass, visit counts and reach probabilities of each player are accumulated, yet all strategies remain fixed. (By contrast, in CFR, the strategy at a node is updated with each CFR visit.) After all visits are counted and probabilities are accumulated, a backward pass computes utilities and updates regrets. FSICFR computational time complexity is proportional to the number of *nodes* times the average node outdegree, whereas CFR complexity is proportional to the number of *node visits* times the average node outdegree. Since the number of node visits grows exponentially in such abstracted problems, FSICFR has exponential savings in computational time per training iteration relative to CFR.

We now present the FSICFR algorithm for two-player, zero-sum games in detail as Algorithm 2. Each player node n consists of a number of fields:

$visits$ - the sum of possible paths to this node during a single training iteration.
$pSum_i$ - the sum of the probabilities that player i would play to this node along each possible path during a single training iteration.
r - a mapping from each possible action to the average counterfactual regret for not choosing that action.
σ - a node strategy, i.e., a probability distribution function mapping each possible action to the probability of choosing that action.
σSum - the sum of the strategies used over each training iteration the node is visited.
$player$ - the player currently taking action.
T - the sum of $visits$ across all training iterations.
v - the expected game value for the current player at that node.

A significant, domain-specific assumption is required for this straightforward, zero-sum form of FSICFR. There is a one-to-one correspondence between player nodes and abstracted information sets, and our visits to nodes must contain enough state information (e.g., predetermined public and private information for both players) such that the appropriate successor nodes may be chosen. In the case of Dudo, this means that the algorithm, having predetermined player rolls, knows which player information sets are legal successors.

We note that the predetermination of chance node outcomes may present difficulties for some games or game abstractions where constraints on a chance node are dependent on the path by which it is reached (e.g., drawing the next card without knowing how many and thus which cards have been previously drawn). This does not pose a problem for Dudo or non-draw forms of Poker. Observe that such predetermination should proceed in topological order, as the predetermination of chance nodes affects the reachability of later chance nodes.

Also, we note that if, as in the case of Dudo, the number of visits to a node will be constant across all training iterations, then one can eliminate the $n.visits$ variable and replace it by the value 1 in equations.

In summary, consider FSICFR as similar to the case where we hold the strategy before a CFR training iteration fixed and execute an entire iteration of regret updates without changing the strategy until after the iteration. In such a case, all operations at a node are the same, and we transform the exponential tree recursion of CFR into a linear dynamic-programming graph-traversal for FSICFR. As a consequence, the node update frequency is equalized rather than exponentially proportional to depth.

4 Experimental Results

The performance of FSICFR did exceed that of CFR for small numbers of dice in play. However, this was also *necessary* for the approximation of an optimal strategy for the full 5-versus-5 game of Dudo. A single iteration of standard CFR became prohibitively expensive for a practical Dudo training even for a small number of dice. In this section, we will begin with a number of experiments (1) to demonstrate a real-time learning performance for the small 2-versus-2 game, (2) to compare iteration costs for both algorithms as we scale to more dice, and (3) to see how an imperfect recall of actions performs when recall is varied[5].

4.1 Learning Outperformance

In order to understand the relative real-time strength of FSICFR versus CFR, we suspend the training threads at specific intervals of seconds (1, 2, 4, 8, etc.), exporting the current strategy and resuming training. CFR defines the strategy for unvisited player nodes as the default uniform strategy.

With a claim history memory limit of 3, we first sought to compare the performance for 1-vs.-1 die training, by comparing a learned strategy versus an optimal strategy. However, within the first second, both algorithms had largely converged, yielding little contrast. We therefore turned our attention to 2-vs.-2 die training. Such a training relies upon precomputed results of 1-vs.-2 and 2-vs.-1 dice training, both of which rely upon 1-vs.-1 die training. For a fair comparison, we used the result of 2 million iterations of FSICFR for these 2- and 3-dice strategies, giving both the same smaller-case training. For each exported strategy, we played 1 million games divided into two identical sets of 500K games with the two players trading positions for each set.

The observed win rates with respect to logarithmic seconds are shown in Figure 3. The contrast in the relative speeds of the two algorithms was so great that CFR training still had unvisited nodes and made some use of the default random strategy through 128 seconds, whereas FSICFR training had visited all nodes before the first export at 1 second. We can see that FSICFR starts with a significant advantage, peaking at an 86.8% win rate at 4 seconds, after which CFR training gradually decreases that edge.

[5] All experiments in this section were performed on Dell Optiplex 990 computers with an Intel Core i7-2600 (3.4 GHz) and 8GB RAM running Ubuntu 11.04.

Algorithm 2. FSICFR(L)

Require: A topologically sorted list L of extensive game DAG nodes.
Ensure: The normalized strategy sum $n.\sigma Sum$, i.e. the average strategy for each player node n, approximates a Nash equilibrium.

1: **for** each training iteration **do**
2: Predetermine the chance outcomes at all reachable chance nodes in L.
3: **for** each node n in order of topologically-sorted, reachable subset $L' \subset L$ **do**
4: **if** n is the initial node **then**
5: $n.visits \leftarrow n.pSum_1 \leftarrow n.pSum_2 \leftarrow 1$
6: **end if**
7: **if** n is a player node **then**
8: Compute strategy $n.\sigma$ according to Equation 5.
9: **if** $n.r$ has no positive components **then**
10: Let $n.\sigma$ be the uniform distribution.
11: **end if**
12: $n.\sigma Sum \leftarrow n.\sigma Sum + (n.\sigma \cdot n.pSum_1$ **if** $n.player = 1$ **else** $n.\sigma \cdot n.pSum_2)$
13: **for** each action a **do**
14: Let c be the associated child of taking action a in n with probability $n.\sigma(a)$.
15: $c.visits \leftarrow c.visits + n.visits$
16: $c.pSum_1 \leftarrow c.pSum_1 + (n.\sigma(a) \cdot n.pSum_1$ **if** $n.player = 1$ **else** $n.pSum_1)$
17: $c.pSum_2 \leftarrow c.pSum_2 + (n.pSum_2$ **if** $n.player = 1$ **else** $n.\sigma(a) \cdot n.pSum_2)$
18: **end for**
19: **else if** n is a chance node **then**
20: Let c be the associated child of the predetermined chance outcome for n.
21: $c.visits \leftarrow c.visits + n.visits$
22: $c.pSum_1 \leftarrow c.pSum_1 + n.pSum_1$
23: $c.pSum_2 \leftarrow c.pSum_2 + n.pSum_2$
24: **end if**
25: **end for**
26: **for** each node n in reverse order of topologically-sorted, reachable subset $L' \subset L$ **do**
27: **if** n is a player node **then**
28: $n.v \leftarrow 0$
29: **for** each action a **do**
30: Let c be the associated child of taking action a in n with probability $n.\sigma(a)$.
31: $n.v(a) \leftarrow (c.v$ **if** $n.player = c.player$ **else** $-c.v)$
32: $n.v \leftarrow n.v + n.\sigma(a) \cdot n.v(a)$
33: **end for**
34: Counterfactual probability $cfp \leftarrow (n.pSum_2$ **if** $n.player = 1$ **else** $n.pSum_1)$
35: **for** each action a **do**
36: $n.r(a) \leftarrow \frac{1}{n.T + n.visits}(n.T \cdot n.r(a) + n.visits \cdot cfp \cdot (n.v(a) - n.v))$
37: **end for**
38: $n.T \leftarrow n.T + n.visits$
39: **else if** n is a chance node **then**
40: Let $[n.player, n.v] \leftarrow [c.player, c.v]$, where c is the predetermined child of n.
41: **else if** n is a terminal node **then**
42: $n.v \leftarrow$ the utility of n for current player $n.player$.
43: **end if**
44: $n.visits \leftarrow n.pSum_1 \leftarrow n.pSum_2 \leftarrow 0$
45: **end for**
46: **end for**

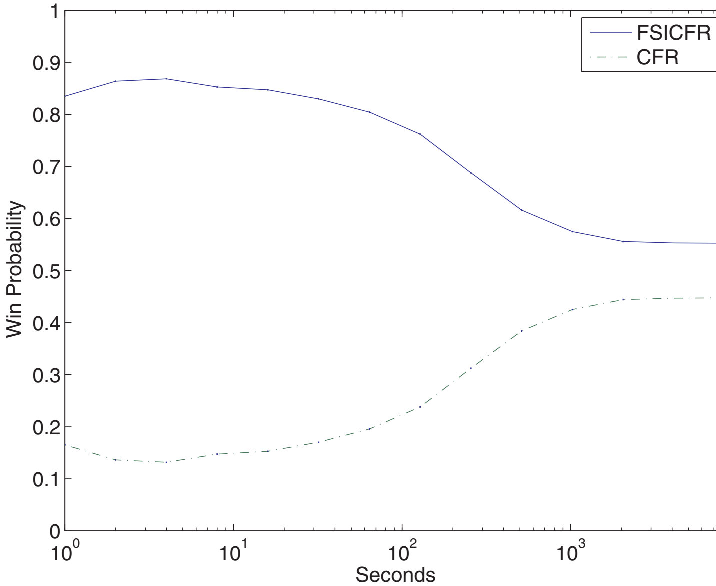


Fig. 3. FSICFR vs. CFR win rates during 2-vs.-2 dice training

Since both algorithms rely on the same update equations, they eventually converge to the same game value. However, CFR's convergence is slow even for 2-vs.-2 dice and, after 8192 seconds of training, still wins only 44.7% of games against FSICFR[6].

4.2 Computational Time Per Training Iteration

The full recursive traversal of CFR causes the relatively slow computation of training iterations and underperformance against FSICFR. Furthermore, when we increase the problem size, FSICFR becomes completely necessary. Consider the time cost per training iteration of FSICFR (Figure 4(a)) and CFR (Figure 4(b)). Entries are omitted in Figure 4(b) where a single training iteration for a single pair of die rolls takes more than four days of computation[7].

Whereas the time cost per FSICFR iteration roughly doubles with each added die in play, the standard CFR cost increases by almost two orders of magnitude for each added die. Thus, for more dice in play, FSICFR not only outperforms CFR, but also converges for all combinations of player/opponent dice before CFR can complete a *single* training iteration for a *single* simulated pair of dice rolls with 7 or more dice. Computation of a Dudo full-game strategy is not feasible with the standard CFR. FSICFR makes feasible (1) the Dudo strategy computation, and (2) the strategy computation for similarly structured games with exponentially growing paths to nodes of greater depth.

[6] Wilson score intervals were used to compute 90% confidence intervals, but such intervals are so small for 1 million games that they are not easily seen in Figure 3.

[7] E.g., a single iteration of 2-vs.-5 dice CFR required $389,244,980$ ms ≈ 4.5 days.

	Opponent Dice				
Dice	**1**	**2**	**3**	**4**	**5**
1	0.3	1.0	2.1	4.9	9.9
2	0.7	2.0	4.7	9.7	18.7
3	2.1	4.7	9.2	18.6	34.4
4	5.0	9.8	18.7	32.4	55.3
5	10.0	18.9	34.8	56.1	94.5

(a) Time (ms) per FSICFR training iteration averaged over 1000 training iterations.

	Opponent Dice				
Dice	**1**	**2**	**3**	**4**	**5**
1	13	49	1551	97210	6179541
2	26	1507	99019	6310784	-
3	1509	98483	6211111	-	-
4	97611	6265326	-	-	-
5	6290658	-	-	-	-

(b) Time (ms) per single CFR training iteration.

Fig. 4. Time (ms) per training iteration

4.3 Varying Imperfect Action Recall

Finally, we turn our attention to the real-time training performance of FSICFR when we vary the imperfect action recall limit. As in earlier experiments, we suspend training threads and export strategies at intervals. In this experiment, however, we train for the simplest 1-vs.-1 die round and compare performance against a known optimal 1-vs.-1 die strategy computed by Todd Neller and Megan Knauss in 2007 using the technique of [4]. The results are shown in Figure 5.

First, we observe that by 512 seconds all training for memories of 2 or more claims converges within 0.15% of the optimal performance. Only limiting the recall to a single claim shows a poor, erratic performance. Smaller claim memory limits yield smaller extensive game trees and converge more quickly with, in general, lesser performance.

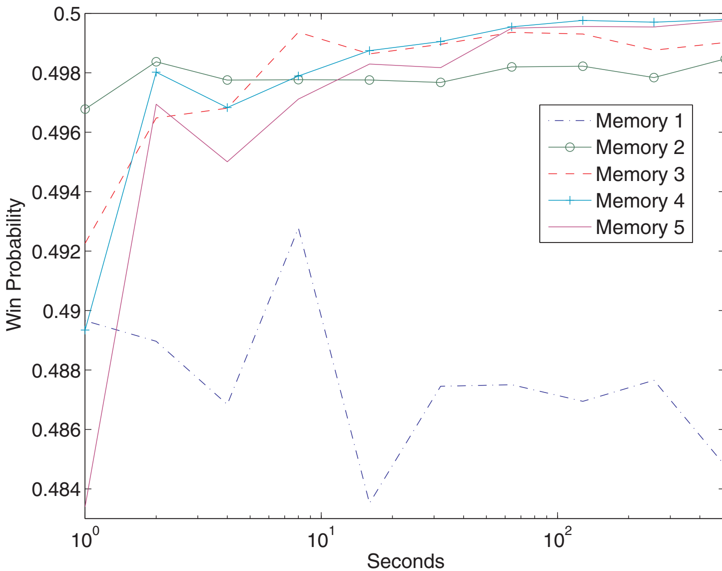


Fig. 5. 1-vs.-1 die FSICFR win rates vs. optimal strategy varying action recall imperfection

However, the eventual superior gain in performance is not so significant for the larger claim memory limits. Thus, a claim memory limit of 3 appears to be a reasonable abstraction for high-quality Dudo play that approximates optimal strategy.

Finally, we note that neither CFR nor FSICFR are guaranteed to converge to a Nash equilibrium when applied to imperfect recall abstractions. Such abstraction pathologies have been studied in [8]. However, experimental evidence (see, e.g., Figure 5) indicates no such pathological behavior for the application of FSICFR to our imperfect recall abstraction of Dudo.

5 Conclusion

In this work, we have computed the first approximation of optimal play for the enduringly popular 15th-century Inca dice game of Dudo. Abstracting the game through an imperfect recall of actions, we reduced the number of information sets from over 2.9×10^{20} to under 2.2×10^{7}. However, Counterfactual Regret Minimization (CFR), a powerful recent technique for competitive computer Texas Hold'em Poker, proved impractical for any application due to the exponentially growing recursive path traversals, even in the abstracted game.

Our primary contribution is the observation that, in such games, the exponential recursive traversal of CFR can be restructured as a polynomial dynamic-programming traversal if we hold the strategies fixed at each node across the entire forward traversal, and update the regrets and strategies only once per node at the concluding backpropagation of utilities. Fixed-Strategy Iteration Counterfactual Regret Minimization (FSICFR) proved not only to be superior in learning rate for a small number of dice in play, but also necessary for the practical computation of an approximate Nash equilibrium for the full 2-player game starting with 5-vs.-5 dice.

At the end of this contribution we conclude by four open questions. (1) What is the true game value for the full game of Dudo?(2) What is the quality of our full-game approximation? (3) What is the best parallel version of this algorithm? (4) Given an approximation of optimal policy, what is the best way to compress the policy so as to minimize the loss of performance? While we have taken significant steps forward, we fully recognize that many interesting questions may lie ahead.

Acknowledgments. The authors would like to thank Marc Lanctot for his great assistance in presenting the overview of CFR, and Megan Knauss for her contribution to the first computation of optimal 1-vs.-1 die Dudo strategy.

References

1. Hart, S., Mas-Colell, A.: A simple adaptive procedure leading to correlated equilibrium. Econometrica 68(5), 1127–1150 (2000)
2. Jacobs, G.: The World's Best Dice Games, new edn. John N. Hansen Co., Inc., Milbrae (1993)
3. Knizia, R.: Dice Games Properly Explained. Elliot Right-Way Books, Brighton Road, Lower Kingswood, Tadworth, Surrey, KT20 6TD U.K (1999)

4. Koller, D., Megiddo, N., von Stengel, B.: Fast algorithms for finding randomized strategies in game trees. In: Proceedings of the 26th ACM Symposium on Theory of Computing (STOC 1994), pp. 750–759 (1994)
5. Lanctot, M., Waugh, K., Zinkevich, M., Bowling, M.: Monte carlo sampling for regret minimization in extensive games. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C.K.I., Culotta, A. (eds.) Advances in Neural Information Processing Systems 22, pp. 1078–1086. MIT Press (2009)
6. Mohr, M.S.: The New Games Treasury. Houghton Mifflin, Boston (1993)
7. Risk, N.A., Szafron, D.: Using counterfactual regret minimization to create competitive multiplayer poker agents. In: Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, vol. 1, pp. 159–166 (2010), http://portal.acm.org/citation.cfm?id=1838206.1838229
8. Waugh, K., Schnizlein, D., Bowling, M.H., Szafron, D.: Abstraction pathologies in extensive games. In: Sierra, C., Castelfranchi, C., Decker, K.S., Sichman, J.S. (eds.) AAMAS (2). pp. 781–788. IFAAMAS (2009)
9. Waugh, K., Zinkevich, M., Johanson, M., Kan, M., Schnizlein, D., Bowling, M.H.: A practical use of imperfect recall. In: Bulitko, V., Beck, J.C. (eds.) SARA. AAAI (2009)
10. Zinkevich, M., Johanson, M., Bowling, M., Piccione, C.: Regret minimization in games with incomplete information. In: Platt, J., Koller, D., Singer, Y., Roweis, S. (eds.) Advances in Neural Information Processing Systems 20, pp. 1729–1736. MIT Press, Cambridge (2008)

The Global Landscape of Objective Functions for the Optimization of Shogi Piece Values with a Game-Tree Search

Kunihito Hoki[1] and Tomoyuki Kaneko[2]

[1] Department of Communication Engineering and Informatics,
The University of Electro-Communications, Tokyo 182-8585, Japan
[2] Department of Graphics and Computer Sciences, The University of Tokyo, Tokyo, Japan
hoki@cs.uec.ac.jp, kaneko@acm.org

Abstract. The landscape of an objective function for supervised learning of evaluation functions is numerically investigated for a limited number of feature variables. Despite the importance of such learning methods, the properties of the objective function are still not well known because of its complicated dependence on millions of tree-search values. This paper shows that the objective function has multiple local minima and the global minimum point indicates reasonable feature values. Moreover, the function is continuous with a practically computable numerical accuracy. However, the function has non-partially differentiable points on the critical boundaries. It is shown that an existing iterative method is able to minimize the functions from random initial values with great stability, but it has the possibility to end up with a non-reasonable local minimum point if the initial random values are far from the desired values. Furthermore, the obtained minimum points are shown to form a funnel structure.

1 Introduction

The heuristic evaluation function is one of the essential elements in game-tree search techniques, and a well-designed evaluation is required in order to build a high-performance tree searcher. A well-known example of designing a heuristic evaluation function can be found in computer chess. In [22], the following chess features are used to make approximate evaluation functions: (1) material balance, (2) pawn formation, (3) board positions of pieces, (4) attacks on a friendly piece to guard, on an enemy piece to exchange, on a square adjacent to the king, and to pin a piece, and (5) mobility. Although more sophisticated features can be found in open source chess programs nowadays [11, 14, 20], these feature selections seem to be an outline for the design of an accurate chess evaluation function. In the literature and chess programs, feature (1) is typically the dominant feature, and the majority of these evaluation functions are hand tuned. It seems that a successful evaluation function in computer chess has been designed, and computer players that use a straightforward evaluation function that works in favor of search speed on the game trees have outperformed the strongest human players [5].

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 184–195, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Despite being successful in chess, this course of action has not always yielded fruitful results with other chess variants. A good evaluation function has proven to be difficult to design for shogi, the Japanese chess variant. Because of the limited mobility of shogi pieces, positional considerations are more important than in chess. Therefore, a massive increase in the number of features has been required for a computer to play a decent game. A technical report [27] describes the evaluation function of a commercial shogi program YSS 7.0, one of the strongest shogi programs in 1997. In this report, the magnitudes of positional feature weights of each shogi piece are comparable to its material value. Because the number of crucial feature weights is too large to adjust manually, machine learning of a shogi evaluation function seems to be an inevitable way to create a strong artificial player.

Machine learning of the heuristic evaluation function is a challenging problem, and researchers have made substantial efforts in various fields related to games (some of them are reviewed in [7]). One of the interesting ideas for learning evaluation functions of chess variants is supervised learning to control the heuristic search so that the search results agree with the desired moves [1, 8, 10, 12, 16, 19, 24]. Several kinds of loss functions that can be formulated as $L(s_j - s_\mathrm{d})$ have been designed in order to guide supervised learning procedures; here, s_d is the score for the desired move, and s_j is the score for another legal move j. Then, the smaller values that these loss functions take, the better the search results will agree with the desired moves. Because the scores of the moves should be determined by game-tree searches, their loss functions are complicated functions of the evaluation feature weights; development of such a supervised learning technique is a challenging goal. Similar treatments, i.e., machine learning in conjunction with tree-searches, can be found in studies on reinforcement learning of evaluation functions [2, 3, 4, 21, 25, 26].

Although there has been only limited success in supervised learning of the heuristic evaluation function, recent computer shogi tournaments have shown signs of success; a fully machine-learned evaluation function performed better than the best human effort of handcrafting [10]. The supervised learning in question employed a numerical minimization of an objective function including a loss function that can be formulated as $L(s_j - s_\mathrm{d})$. We take the term “objective function” to mean a function to be minimized as an optimization problem. The complexity of positional evaluations in shogi means that a large-scale feature vector must be optimized. In fact, the reported method numerically optimized more than 10 million feature weights to a local minimum with reasonable numerical accuracy.

Despite the above-mentioned success, an efficient and stable optimization algorithm ought to be developed to find a better local minimum or to increase the feature variables. It is known that the performance of numerical optimizations is sensitive to the landscape of the objective function [6, 18, 23]. One example of an easy problem is minimization of a convex function; if a local minimum exists, then it is a global minimum. Figure 1 shows the properties of particular functions and their difficulties regarding numerical minimization. Because various global and local convergence algorithms are available, minimization of a smooth function (b) can be thought as an easy problem. In contrast, minimization of non-differentiable functions are often more difficult than cases (a) or (b), because a quadratic model, such as the

Hessian approximation of the conjugated gradient method [18, 23], is not always appropriate for these functions. The function (d) is also a difficult target, because an important local minimum is hidden inside a deep narrow trough, and it is quite difficult to find it by using numerical iteration methods. The last function, the most difficult example, is a non-continuous one; even primitive iterative methods such as gradient decent are generally not capable of finding its minimum.

In the case of optimization of the shogi feature weights, the loss function's dependence on millions of search values makes any analysis of its properties difficult. So far, only some of the properties of this function have been reported, and these assume that the tree search is strictly based on the minimax algorithm [10]: (a) the function is continuous with the feature weights, and (b) the function is not always partially differentiable with the feature weights if the result of the tree search has two or more distinct leaf positions of the principal variations (PVs).

In this paper, we investigate the properties of the objective function without assuming that the game-tree searches are strictly based on the minimax algorithm. In reality, strict minimax values are not available for shogi. The practical searchers use integer alpha and beta variables for the alpha-beta pruning [13]. In addition, forward pruning techniques are required so that supervised learning can be used on current computers, because the loss function $L(s_j - s_\mathrm{d})$ is too complicated for anyone to imagine the landscape of the objective function. To address this difficulty, we visualize this function in two-dimensional planes of piece values. Moreover, we catch a glimpse of the properties of the function with thirteen-dimensional weights; these weights are able to represent all shogi piece values. By using these model cases, we investigate the continuity and partial differentiability of the loss function, as well as the global convergence and locations of local minima found by the optimization procedure in computational experiments.

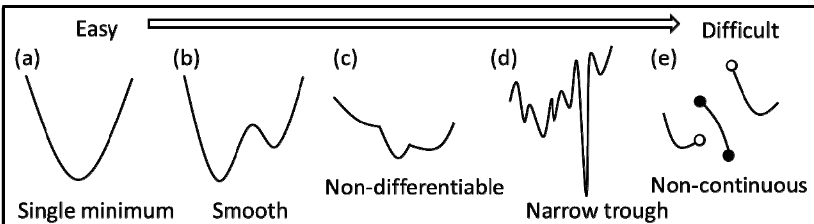


Fig. 1. Example illustrating the difficulties facing any minimization procedure

2 Two-Dimensional Landscape of the Loss Function

Several loss functions $L(s_j - s_\mathrm{d})$ have been designed for supervised learning of evaluation functions [1,8,10,12,16,19,24]. These functions generally use

$$J(\boldsymbol{P},\boldsymbol{v}) = \sum_{p \in \boldsymbol{P}} \sum_{m=2}^{M_p} T_p[\xi(p_m,\boldsymbol{v}) - \xi(p_1,\boldsymbol{v})] \tag{1}$$

Here, $\boldsymbol{P}$ is a set of game positions, M_p is the number of legal moves in p, p_1 is the desired child position of p, $\xi(p,\boldsymbol{v})$ is the search value of a game tree rooted at p, and $\boldsymbol{v}$ is the feature vectors of an evaluation function $f(p,\boldsymbol{v})$. The set $\boldsymbol{P}$ and the desired child positions are often sampled from game scores of human experts.

The loss function is $T_p(x) = T(+x)$ when the player to move in position p is the maximizing player; otherwise, $T_p(x) = T(-x)$. When we set $T(x)$ as the step function, the loss function represents the number of legal moves that are evaluated as being better than the moves actually played in the scores. In this paper, a sigmoid function $T(x) = 1 / (1 + e^{-\alpha x})$ is employed as an approximation of the step function, as is done in [8,10]. α in the sigmoid function was set to 0.0273, so that $T(x)$ varies significantly if x changes by a few hundred.

A few useful properties are shown in [10]. That is, (1) when the search value $\xi(p,\boldsymbol{v})$ is computed based on the minimax algorithm and the evaluation function $f(p,\boldsymbol{v})$ is continuous with respect to $\boldsymbol{v}$, $\xi(p,\boldsymbol{v})$ is also continuous [10], (2) the search value is not always partially differentiable if two or more distinct leaf positions of PVs exist in the tree search rooted at p, and (3) when the search value is partially differentiable, the partial derivative of the search value is

$$\partial\xi(p,\boldsymbol{v})/\partial v_i = \partial f\left(p^{\text{leaf}},\boldsymbol{v}\right)/\partial v_i \ , \tag{2}$$

where p^{leaf} is a unique leaf position.

The evaluation function for shogi piece values is

$$f(p,\boldsymbol{v}) = \sum_{i=1}^{13}\left[n_i(p) - n_i(\tilde{p})\right]v_i \ , \tag{3}$$

where i represents each type of shogi piece, v_i is the value of the i-th type of piece, and $n_i(p)$ is the number of i-th pieces owned by Black in position p. Note that Black (White) refers to the player who plays the first (second) move. $\tilde{p}$ is a complete reversal of black and white sides at position p.

In this section, we investigate the properties of the loss function $J(\boldsymbol{P},v_1,v_2)$ of only two piece values. This limitation allows us to take a look at the global map of the two-dimensional landscape created by practical search values. We construct the loss function $J(\boldsymbol{P},\boldsymbol{v})$ from N = 112,613 positions occurring in the 1,000 game scores from games played between human experts. Here, to avoid redundancy in the game scores, duplications in the N positions were detected by using a hash key for each position. The function values normalized by $A = \sum_{p\in\boldsymbol{P}} (M_p - 1) = 7{,}960{,}015$ are shown in this paper. Note that A represents the maximum value of $J(\boldsymbol{P},\boldsymbol{v})$, where the worst condition of $T_p(x) = 1$ is satisfied for all p_m and p_1 in Eq. (1).

We used a computer shogi program called BONANZA, which employs a shallow search including a quiescence search; the source code is available through the Internet [9]. The program uses conventional techniques such as PVS [15, 17], capture search at the frontier node as a quiescence search, check and capture extensions, transposition tables, static exchange evaluation, killer and history heuristics, null

move pruning, futility pruning, and late move reductions. The nominal depth of the tree search was set to 1-ply for this experiment. Note that this nominal depth is a guiding principle of the search depth; the actual game tree can be deeper along some branches and shallower along others than a uniform 1-ply game tree.

The contour maps of the two-dimensional functions were drawn using sampled function values at an interval of 10 for a pair of variables, and reasonable constant values were assigned to the other piece values, i.e., 118 (pawn), 273 (lance), 318 (knight), 477 (silver general), 562 (gold general), 620 (bishop), 734 (rook), 485 (promoted pawn), 387 (promoted lance), 445 (promoted knight), 343 (promoted silver), 781 (promoted bishop), and 957 (promoted rook). Note that a contour line of the functions is a curve along which the functions take a constant value. The contour lines have certain properties: the gradient of the function is perpendicular to the lines, and when two lines are close together, the magnitude of the gradient is large.

Figure 2 shows an enlarged contour map of $J(\boldsymbol{P}, v_{\text{pawn}}, v_{\text{lance}})$. The lines were drawn with an interval of 0.01. The map was computed with the ranges of [10, 1000] for a pawn and [10, 840] for a lance. The function simply increases and no interesting structure is observed outside of the enlarged map. The single minimum 'X' is at reasonable pawn and lance values. Therefore, machine learning of these two piece values can be achieved by minimizing the function value. Although the search value $\xi(p,\boldsymbol{v})$ in Eq. (1) is a complicated function because of the use of practical measures such as use of forward pruning and integer variables, this figure shows no sudden changes. It means that the function is approximately continuous and amenable to piece value optimization. Gradient decent thus seems to be a feasible way to find the single minimum 'X' of this two-dimensional function.

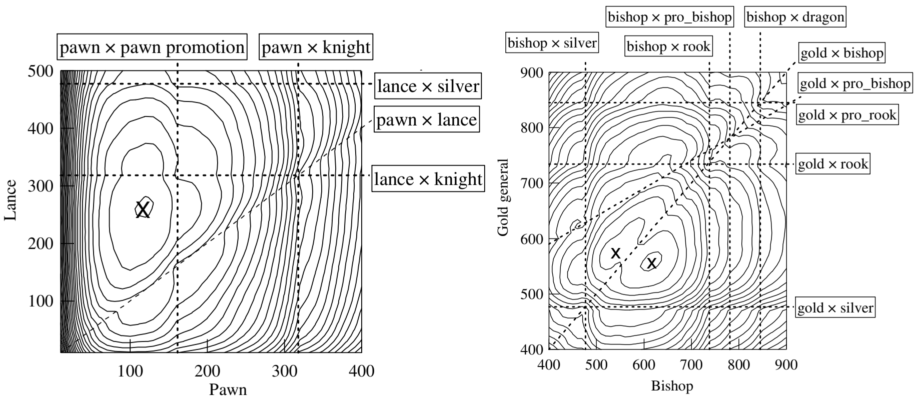


Fig. 2. (Left panel) Enlarged contour map of $J(\boldsymbol{P}, v_{\text{pawn}}, v_{\text{lance}})$. (Right panel) Enlarged contour map of $J(\boldsymbol{P}, v_{\text{gold}}, v_{\text{bishop}})$. The broken lines indicate critical boundaries at which the two-dimensional function is not partially differentiable. The minima are indicated by 'X'.

However, we also see disadvantageous properties, i.e., clear edges of the contour lines. This indicates that the function is not partially differentiable at these points. The broken lines in Fig. 2 are critical boundaries at which the profit and loss ratio of

material exchanges inverts itself. For example, a knight is more valuable than a pawn, but the winning a pawn becomes more profitable than winning a knight when the pawn value is greater than 318. This boundary is labeled in Fig. 2 as "pawn × knight". Also, a case where winning a pawn while allowing the promotion of an opponent's pawn is more profitable when the pawn value is greater than 162. This boundary is labeled in Fig. 2 as "pawn × pawn promotion". Note that the boundary value of the pawn (162) is computed from 2×(pawn value) = promoted-pawn value – pawn value.

However, the loss function is not partially differentiable at these critical boundaries. The reason is that the inversion of profit and loss in the evaluation causes a switch of two or more leaf positions of PVs in Eq. (2), so that the partial-differential value jumps discontinuously across the boundary to a substantially different value. This result indicates that the use of a quadratic model is not appropriate for numerical optimization when the initial guess of $\boldsymbol{v}$ is outside of these boundaries.

These observations tell us that a non-linear evaluation function may create further complexity in the loss function. Because the piece-value dependency on the evaluation function is linear, the corresponding boundaries shown in Fig. 2 are straight lines. If the evaluation function has non-linear dependencies, the two-dimensional function may have more boundary curves.

Furthermore, if the occurrences of gain and loss are homogeneous over all evaluation features, the loss function may be more complicated than in Fig. 2. For instance, there may be more boundaries, e.g., a boundary at which the profit and loss ratio of exchanging a pawn with a promoted lance inverts itself, than in Fig. 2. The reason why these boundaries do not appear in this figure is that the occurrence of these material gains or losses in all PVs of the set $\boldsymbol{P}$ are too rare to see the influence on the function $J(\boldsymbol{P}, v_{\text{pawn}}, v_{\text{lance}})$. For the same reason, there is no a boundary due to exchanges of two or more pieces.

Figure 2 also shows an enlarged contour map of $J(\boldsymbol{P}, v_{\text{gold}}, v_{\text{bishop}})$. The lines are drawn at intervals of 0.005. Here, we can see more critical boundaries and the landscape is more complicated than in the left panel of Fig. 2. A second notable difference is that the function has two local minima on both sides of the "gold × bishop" boundary (denoted by 'x'). The loss function suggests two different pairs of gold-general and bishop values. Because a bishop is more valuable than a gold general in most cases, the global minimum point is reasonable. In contrast, a gold general is more valuable than a bishop in the second lowest minimum point. This means (1) that the optimized result depends on the choice of initial values of the iterative method such as the gradient decent, and (2) that the optimization procedure has some chance of ending up with an unnatural result.

Figure 3 shows an enlarged contour map of $J(\boldsymbol{P}, v_{\text{pro_lance}}, v_{\text{pro_knight}})$. This map has six local minima, and the lowest one corresponds to reasonable piece values. The lines are drawn at intervals of 0.00002. Because promoted lances and promoted knights appear less frequently in game scores and PVs, the interval of the contour-line levels is 500 times smaller than in the left panel of Fig. 2. This means that the map of a promoted lance and a promoted knight has an almost flat surface when the same scaling is used. This property, i.e., the order of function scaling differs from those of

other variables, discourages the use of a naïve gradient decent for optimization of all thirteen piece values. Because the gradient vectors are nearly orthogonal to these flat directions, values of these less frequently appearing pieces do not change sufficiently with each gradient-decent step. This scaling problem has to be solved before the shogi piece values can be fully optimized.

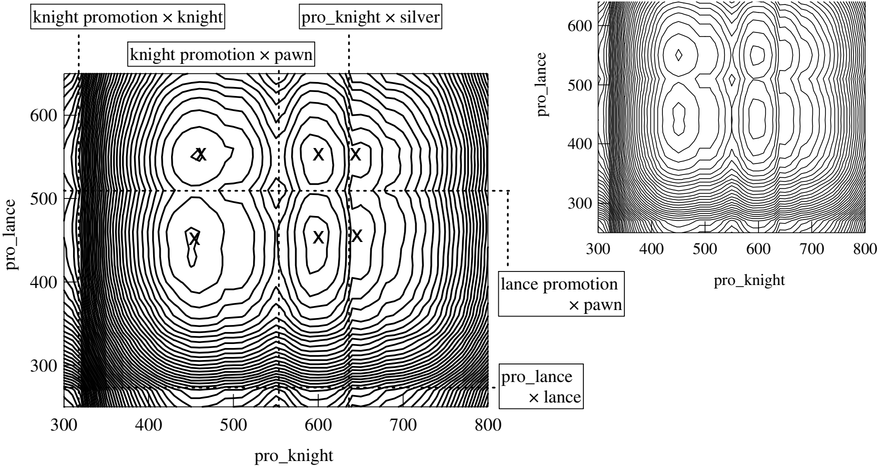


Fig. 3. (Left panel) Enlarged contour map of $J(\boldsymbol{P}, v_{\mathrm{pro_lance}}, v_{\mathrm{pro_knight}})$. The broken lines indicate critical boundary at which the two-dimensional function is not partially differentiable. The six minima are indicated by 'X'. (Right panel) Enlarged contour map of $J_1(\boldsymbol{P}, v_{\mathrm{pro_lance}}) + J_2(\boldsymbol{P}, v_{\mathrm{pro_knight}})$. Here, the two variables are approximately decomposed, and this map looks similar to the one in the left panel.

A second notable property of this two-dimensional function is that the coupling between values of a promoted lance and a promoted knight is rather weak. Therefore, the two-dimensional function can be approximately decomposed into two one-dimensional functions, i.e., $J(\boldsymbol{P}, v_{\mathrm{pro_lance}}, v_{\mathrm{pro_knight}}) \cong J_1(\boldsymbol{P}, v_{\mathrm{pro_lance}}) + J_2(\boldsymbol{P}, v_{\mathrm{pro_knight}})$. The result of this approximation is shown in the right panel of Fig. 3. Here, we can see that the main features are reproduced by this decomposition. Because the effective degrees of freedom of this function are smaller than the actual ones, this property will be advantageous for making a numerical procedure of multi-dimensional optimizations. That is, $v_{\mathrm{pro_lance}}$ and $v_{\mathrm{pro_knight}}$ can be optimized separately as a first approximation.

3 Full Optimization of Shogi Piece Values

In the previous section, we observed two properties of the loss function of two variables. Here, we shall lift the restriction from two variables to thirteen variables, which is sufficiently large to express all piece values in shogi. The aim of this experiment is to catch a glimpse of the global landscape map and numerical global convergences for the full piece values. For this purpose, a Monte Carlo sampling of

the initial guess of the optimization procedures was carried out to enumerate local minima, and the obtained minimum points were then analyzed.

The supervised learning method in [8, 10] employs an objective function including the loss function in Eq. (1):

$$J'(\boldsymbol{P},\boldsymbol{v}) = \sum_{p \in \boldsymbol{P}} \sum_{m=2}^{M_p} T_p \left[\xi(p_m,\boldsymbol{v}) - \xi(p_1,\boldsymbol{v})\right] + \lambda g(\boldsymbol{v}) \tag{4}$$

The second term consists of a Lagrange multiplier λ and an equality constraint $g(\boldsymbol{v})$ = 0. This term stabilizes the optimization procedure by explicitly removing an uncertain positive value a of the scaling $\tilde{\boldsymbol{v}} = a\boldsymbol{v}$. The constraint function is set to $g(\boldsymbol{v})$ = $\left(\sum_{i=1}^{13} v_i\right)$ − 6,500. Here, the magnitude of the constant 6,500 is chosen in accordance with the 16-bit integer representation of the evaluation function in [9]. In these references, the objective function also has a regularization term to prevent overfitting of $\boldsymbol{v}$. However, we did not include this regularization term in our study because the number of feature values was only thirteen.

When the search value is partially differentiable, the partial derivative can be written using Eqs. (2), (3), and (4) as [8, 10]

$$\frac{\partial J'(\boldsymbol{P},\boldsymbol{v})}{\partial v_i} = \sum_{p \in \boldsymbol{P}} \sum_{m=2}^{M_p} \frac{dT_p(x)}{dx} \left[n_i\left(p_m^{\text{leaf}}\right) - n_i\left(\tilde{p}_m^{\text{leaf}}\right) - n_i\left(p_1^{\text{leaf}}\right) + n_i\left(\tilde{p}_1^{\text{leaf}}\right) \right] + \lambda \frac{\partial g(\boldsymbol{v})}{\partial v_i} . \tag{5}$$

where p^{leaf} is the unique leaf position of a corresponding principal variation (PV) of a tree search rooted at p. Starting from an initial guess of $\boldsymbol{v}$ that satisfies the equality constraint $g(\boldsymbol{v})$ = 0, the numerical optimization can be carried out using the following updates for all weights [8, 10],

$$v_i^{\text{new}} = v_i^{\text{old}} - h\,\text{sgn}\left[\partial J'\left(\boldsymbol{P},\boldsymbol{v}^{\text{old}}\right) / \partial v_i\right]. \tag{6}$$

Here, the sign function avoids the scaling problem described in Sec. 2, and h is an integer step size. In this paper, the smallest integer step h = 1 is used. The value of the Lagrange multiplier λ in Eq. (5) is computed every step to keep the equality constraint $g(\boldsymbol{v})$ = 0. Because the bottleneck in computation is the shallow tree searches to obtain a set of p_m^{leaf} in Eq. (5), the set was computed every 32 steps. We counted the number of iterations in the leaf set computation.

The nominal search depth was extended to 2-ply to see whether these optimization steps are stable with deeper searches. The initial values of the shogi pieces were calculated from pseudo-random numbers. Here, a uniform random integer in the range of [0, 32767] was assigned to each piece value, and the feature vector is scaled as $\tilde{\boldsymbol{v}} = a\boldsymbol{v}$ to satisfy the equality condition of $g(\boldsymbol{v})$ = 0.

Figure 4 shows the results of 78 runs of optimizations. Starting from 78 sets of random initial piece values, a large number of function values numerically converged in 30 iterations. Here, the numerical convergences can be verified by observing the slight oscillation of the function values due to the finite step size.

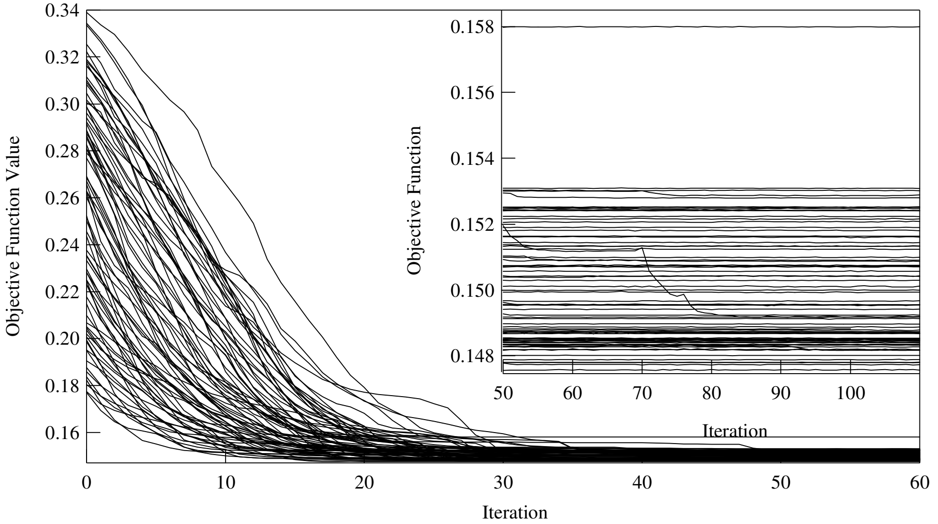


Fig. 4. Results of 78 runs of optimizing the values of thirteen pieces. Here, the initial piece values were determined from pseudo-random numbers. The inset is an enlargement showing the numerical convergences. The ordinate shows the value of the objective function in Eq. (4).

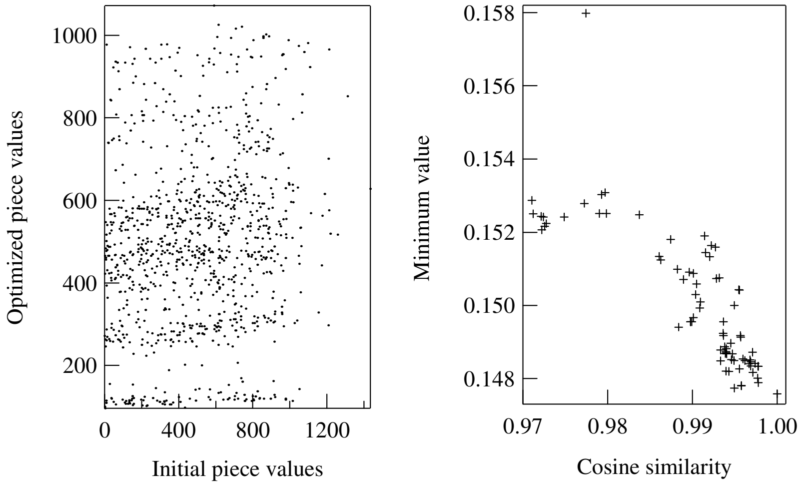


Fig. 5. (Left panel) Scatter plot for 78 sets of optimized piece values with initial piece values. The number of a set of piece values in shogi is thirteen, so 13×78 points are plotted. (Right panel) Scatter plot of the minimum value of the objective function with the cosine similarity between the optimized and best piece values. 78 sets are plotted.

Table 1. Two sets of piece values with the best and worst minimum values of 78 sets. The gray column values are for promoted pieces

	pawn	lance	knight	silver	gold	bishop	rook	silver	lance	knight	silver	bishop	rook
best	105	227	279	444	529	576	700	483	348	535	543	730	1002
worst	206	204	190	470	577	629	760	511	379	152	584	813	1025

In Fig. 4, we see that almost all of the optimization runs end up with different local minima. Here, the learned result shows a certain tendency. That is, a lower minimum value means a better minimum point. As shown in Table 1, the global minimum point with the value 0.1476 (see Fig. 4) gives reasonable piece values. In contrast, the highest minimum point with the value 0.1580 gives unreasonable piece values; the pawn value is too large, and the promoted-knight value is too small.

As shown in the left panel of Fig. 5, there is no significant relation between the initial random values and the corresponding optimized values; the correlation coefficient is 0.15. This result indicates that this optimization method adjusted all of the initial piece values by a sufficient amount. Figure 5 also shows a second plot of the minimized function values with cosine similarities between the optimized and the best piece values. All minimum values lower than 0.148 exist at a location with a similarity higher than 0.993. This indicates that the minimum points form a funnel structure. That is, the closer the minimum point is to the best piece values, the smaller the minimum value is.

4 Conclusions

We investigated the properties of the objective function for supervised learning of the shogi evaluation function. The landscape of this function is hard to imagine, because the function value depends on millions of search values. Furthermore, the search values should be computed by a practical tree searcher that employs forward pruning techniques and integer alpha and beta values. Therefore, by using a model case where the set of feature weights is limited to piece values, the global landscape of this function could be revealed in computational experiments.

The two-dimensional contour maps showed that the global minimum point indicates reasonable piece values. The maps also showed continuous but non-partially differentiable surfaces. The locations of the non-partially differentiable points were at the critical boundaries between profits and losses in piece exchanges. These results indicate that use of quadratic models such as the Hessian approximation of the conjugated gradient method is not appropriate for the numerical optimization when the initial guess of the feature weights is outside of the critical boundaries. From these observations, the expected shape of a one-dimensional cross section of the objective function to optimize shogi piece values can be qualitatively explained by Fig. 1c.

It was also shown that a piece that appears less frequently in the game score has less influence on the other piece variables. Using this property, the two-dimensional function can be decomposed into two one-dimensional functions. This approximation was shown for the map of a promoted lance and a promoted knight. It was also shown

that inhomogeneous appearance frequencies of pieces in the game scores cause a scaling problem. In fact, the contour map of a promoted lance and a promoted knight looks almost flat when the scale of the contour levels is set to that of a pawn and a lance. This scaling problem discourages the use of naïve gradient decent for numerical optimization [18].

A property relating to global convergence of the iterative optimization method presented in [8, 10] was numerically tested by using 78 randomly prepared sets of initial piece values. Here, all thirteen piece values were optimized. Although this method has no guarantee of converging to a solution from arbitrary initial values, reasonable stable convergences for all 78 sets of random initial values were obtained. Furthermore, all of the initial piece values were adjusted by a sufficient amount, and the minimum points were sufficiently better than the random piece values. These results indicate that the optimization method in [8, 10] is promising for minimizing the loss function analyzed in Sec. 2.

However, almost all of the 78 optimizations ended up with different local minima. These minimum points showed a funnel structure. That is, the closer to the best piece values a minimum point is, the smaller the minimum value is. These results indicate that initial piece values that are at least better than random values are desirable when the goal is to learn a reasonable piece value.

In this paper, we analyzed the properties of the objective function for only piece values, and optimized the function. The numerical optimization of full evaluation features including positional considerations would be a very interesting goal, and we would like to handle it in our future work.

References

1. Anantharaman, T.: Evaluation tuning for computer chess: Linear discriminant methods. ICCA Journal 20, 224–242 (1997)
2. Baxter, J., Tridgell, A., Weaver, L.: TDLeaf(λ) Combining temporal difference learning with game-tree search. In: Proceedings of the 9th Australian Conference on Neural Networks (ACNN 1998), Brisbane, Australia, pp. 168–172 (1999)
3. Baxter, J., Tridgell, A., Weaver, L.: Learning to play chess using temporal-differences. Machine Learning 40, 242–263 (2000)
4. Beal, D.F., Smith, M.C.: Temporal difference learning applied to game playing and the results of application to shogi. Theoretical Computer Science 252, 105–119 (2001)
5. Campbell, M., Joseph Hoane, J.A., Hsu, F.: Deep Blue. Artificial Intelligence 134, 57–83 (2002)
6. Conn, A.R., Scheinberg, K., Vicente, L.N.: Introduction to Derivative-Free Optimization. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics (SIAM), Philadelphia (2009)
7. Fürnkranz, J.: Machine Learning in Games: A Survey. In: Fürnkranz, J., Kubat, M. (eds.) Machines that Learn to Play Games, pp. 11–59. Nova Science Publishers (2001)
8. Hoki, K., Kaneko, T.: Large-Scale Optimization of Evaluation Functions with Minimax Search (in preparation)
9. Hoki, K.: Bonanza – The Computer Shogi Program (2011) (in Japanese), `http://www.geocities.jp/bonanzashogi/` (last access: 2011)

10. Hoki, K.: Optimal control of minimax search results to learn positional evaluation. In: Proceedings of the 11th Game Programming Workshop (GPW 2006), Hakone, Japan, pp. 78–83 (2006) (in Japanese)
11. Hyatt, R.: Crafty 23.4 (2010), `ftp://ftp.cis.uab.edu/pub/hyatt`
12. Kaneko, T.: Learning evaluation functions by comparison of sibling nodes. In: Proceedings of the 12th Game Programming Workshop (GPW 2007), Hakone, Japan, pp. 9–16 (2007) (in Japanese)
13. Knuth, D.E., Moor, R.W.: An Analysis of Alpha-Beta Pruning. Artificial Intelligence 13, 293–326 (1991)
14. Letouzey, F.: Fruit 2.1 (2005), `http://arctrix.com/nas/chess/fruit`
15. Marsland, T., Campbell, M.: Parallel Search of Strongly Ordered Game Trees. ACM Computing Survey 14, 533–551 (1982)
16. Marsland, T.A.: Evaluation-Function Factors. ICCA Journal 8, 47–57 (1985)
17. Marsland, T.A., Member, S., Popowich, F.: Parallel game-tree search. IEEE Transactions on Pattern Analysis and Machine Intelligence 7, 442–452 (1985)
18. Nocedal, J., Wright, S.: Numerical Optimization. Springer (2006)
19. Nowatzyk, A.: (2000), `http://tim-mann.org/DTevaltune.txt` (last access: 2010)
20. Romstad, T.: Stockfish 1.9.1 (2010), `http://www.stockfishchess.com`
21. Schaeffer, J., Hlynka, M., Jussila, V.: Temporal difference learning applied to a high-performance game-playing program. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 529–534. Morgan Kaufmann Publishers Inc., San Francisco (2001)
22. Shannon, C.E.: Programming a Computer for Playing Chess. Philosophical Magazine, Ser. 7 41(314) (1950)
23. Sun, W., Yuan, Y.-X.: Optimization Theory and Methods. Nonlinear Programming. Springer Science+Business Media, LLC (2006)
24. Tesauro, G.: Comparison training of chess evaluation functions. In: Furnkranz, J., Kumbat, M. (eds.) Machines that Learn to Play Games, pp. 117–130. Nova Science Publishers (2001)
25. Tesauro, G.: Programming backgammon using self-teaching neural nets. Artificial Intelligence 134, 181–199 (2002)
26. Veness, J., Silver, D., Uther, W., Blair, A.: Bootstrapping from game tree search. In: Bengio, Y., Schuurmans, D., Laerty, J., Williams, C.K.I., Culotta, A. (eds.) Advances in Neural Information Processing Systems 22: 23rd Annual Conference on Neural Information Processing Systems, Vancouver, BC, Canada, pp. 1937–1945 (2009)
27. Yamashita, H.: YSS 7.0 – data structures and algorithms (in Japanese), `http://www32.ocn.ne.jp/~yss/book.html` (last access: 2010)

Solving BREAKTHROUGH with Race Patterns and Job-Level Proof Number Search

Abdallah Saffidine[1], Nicolas Jouandeau[2], and Tristan Cazenave[1]

[1] LAMSADE, Université Paris-Dauphine
[2] LIASD, Université Paris 8

Abstract. BREAKTHROUGH is a recent race-based board game usually played on a 8×8 board. We describe a method to solve 6×5 boards based on (1) race patterns and (2) an extension of Job-Level Proof Number Search (JLPNS).

Using race patterns is a new domain-specific technique that allows early endgame detection. The patterns we use enable us to prune positions safely and statically as far as 7 moves from the end.

For the purpose of solving BREAKTHROUGH we also present an extension of the parallel algorithm JLPNS, viz. when a PN search is used as the underlying job. In this extension, partial results are regularly sent by the clients to the server.

1 Introduction

In this paper, we address the use of parallelization to solve games. We use the game BREAKTHROUGH [10] as a testbed for our experiments with parallel solving algorithms.

BREAKTHROUGH has already been used as a testbed in other work [20,9]. It is an interesting game which offers new challenges to the AI community. We therefore also try to improve on domain-specific techniques. To this effect, we present the idea of *race patterns*, a new kind of static patterns that allow to detect a win several moves before the actual game ends. The use of race patterns has some links with the use of threats when solving GO-MOKU [1]. However, the threats used in GO-MOKU by Allis were designed to select a small number of moves to search, whereas the race patterns are designed to stop the search early.

Research on parallel game-tree search was initially mainly about the parallelization of the Alpha-Beta algorithm. A survey on the parallelization of Alpha-Beta can be found in Mark Brockington's PhD thesis [4]. Other sources about the use of transposition tables in parallel game-tree search and Alpha-Beta are Feldmann et al.'s paper [8] and Kishimoto and Schaeffer's paper [12].

More recently, the work on the parallelization of game-tree search algorithms has addressed the parallelization of Monte-Carlo Tree Search algorithms [5,6,7]

Other related works deal with the parallelization of PDS [14,13] and of Depth First Proof Number search (DF-PN) [11]. A technique to reduce the memory usage of DF-PN is the garbage collection of solved trees [15].

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 196–207, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Previous attempts at parallelizing the Proof Number Search (PNS) algorithm used randomization [16] or a specialized algorithm called at the leaves of the main search tree [21].

Proof-Number search and parallel algorithms were also already successfully used in solving Checkers [18,19].

In this paper we focus on the parallelization of the PN^2 algorithm. The PN^2 algorithm has enabled to solve complex games such as FANORONA [17]. Our goal is to solve such games faster with a similar but parallel algorithm.

The second section is about PNS, Job-Level Proof Number Search (JLPNS) and our algorithm Parallel PN^2 (PPN^2). The third section deals with race patterns at BREAKTHROUGH and the fourth section details experimental results. Finally, section five contains a discussion and a conclusion.

2 Job-Level Proof Number Search

In this section we start presenting PNS (2.1). Then we recall the parallelization of PNS with Job-Level parallelization (2.2). The third section (2.3) presents our Parallel PN^2 algorithm.

2.1 Proof Number Search

PNS was proposed by Allis et al. [2] The goal of the algorithm is to solve sequential perfect information games. Starting from the root position, it develops a tree in a best first manner. PNS uses the concept of effort numbers to compare leaves.

Effort numbers are associated to nodes in the search tree and try to quantify the progress made towards some goal. Specifically in PNS, two effort numbers are used: (1) the proof number PN of a node n estimates the remaining effort to prove that n is winning for Max, and (2) the disproof number DN estimates the remaining effort to prove a win for Min. Originally, the PN (respectively the DN) of a node n was a lower bound on the number of node expansions needed below n to prove that n is a Max win (respectively a Max loss). When the PN reaches 0 (respectively ∞), the DN reaches ∞ (respectively 0), and the node has been proved to be a Max win (respectively a Max loss).

The PN and DN are recursively defined as shown in Table 1 where Win (respectively Lose) designate a terminal node corresponding to a position won by Max (respectively Min), Frontier designate a non-expanded non-terminal leaf node. *Max* (respectively *Min*) designate an expanded internal node with Max (respectively Min) to play.

To select which node to expand next, Allis et al. defined the set of *most proving nodes* [2] and showed that it is possible to select one of them by the following descent procedure. Iterate until a Frontier node is reached: when at a *Max* node, select a child minimizing PN; when at a *Min* node, select a child minimizing DN.

Table 1. Determination of effort numbers for PNS

Node type	PN	DN
Win	0	∞
Lose	∞	0
Frontier	1	1
Max	$\min_{c \in \text{chil}(n)} \text{PN}(c)$	$\sum_{c \in \text{chil}(n)} \text{DN}(c)$
Min	$\sum_{c \in \text{chil}(n)} \text{PN}(c)$	$\min_{c \in \text{chil}(n)} \text{DN}(c)$

2.2 Job-Level Parallelization

Job-Level Proof Number Search [21] has been used to solve CONNECT6 positions. The principle is to have a main Proof Number Search tree, but instead of having plain leaves, a solver is called at each leaf in order to evaluate it.

In order to avoid having several clients trying to prove the same leaf, JLPNS uses a virtual-loss mechanism.[1] When a leaf is sent to a client, it is temporarily assumed to be proved a loss until the client returns a meaningful result.

A disadvantage of the virtual-loss mechanism is that it is possible for a node to be considered losing for some time, but then to be updated to a non-solved state. Stating this otherwise, 0 and ∞ are no longer attractor values for the proof and disproof numbers.

An advantage of the approach taken by JLPNS is that it allows an easy parallelization over a distributed system with a very small communication overhead.

2.3 Parallel PN^2

The principle of the PN^2 algorithm [1,3] is to develop another PN search at each leaf of the main PN search tree in order to have more informed proof and disproof numbers. For PPN^2 search, the PN search tree at the leaves is developed on a remote client. There are at least three differences between PPN^2 and JLPNS.

A first difference is that the PPN^2 algorithm that is called at the leaves is also a Proof Number Search instead of a specialized solver as in JLPNS. As a result, partial results from the unfinished remote search in one client can be sent back to the server to update the main PNS tree in order to influence the next searches of the other clients.

A second difference is that we do not use the virtual loss mechanism to avoid currently computed leaves but a flag on these leaves. First, in our technique, a node is never considered to be losing unless it has actually been proved to be losing, thus 0 and ∞ remain attractors. Then, noting that the set of most-proving nodes usually contains several nodes, our technique ensures that we will pick tasks from the set of most-proving nodes of the current tree. Finally, the

[1] The authors of JLPNS also tried a virtual-win policy and a greedy mechanism which are conceptually similar to virtual-loss [21].

virtual-loss mechanism does not fit well with the partial result update described in the preceding paragraph.

A third difference is that, our algorithm also has BREAKTHROUGH-specific knowledge: it uses the mobility heuristic and race patterns defined in Section 3.3.

Just as in PN^2, the size of the remote tree can either be fixed or a function of the size of the main search tree.

The main algorithm which is run on the server, is described in Algorithm 1. It consists in receiving results from the clients and updating the tree according to these results. A result can either be a partial result or a final result. In both cases, we need to update the proof and disproof numbers of the concerned leaf with the result. We also update recursively its ancesters. When the result is final, however, we expand the tree and need to find a new not reserved leaf for the now idle client. Finding a not reserved leaf is done by a backtracking algorithm where the choice points are the nodes with several children minimizing the proof or disproof number.

Algorithm 1. Main algorithm.

```
while root is not proved do
  receive result r from any client c
  if r is a partial result then
    update the PN and DN with r
  else
    expand the tree
    update the PN and DN with r
    if root is proved then
      break
    end if
    find the most proving and not reserved leaf l
    reserve leaf l
    send the position at l to client c
  end if
end while
collect and discard the remaining client messages
send stop to all clients
```

The remote algorithm which is run on the clients is described in Algorithm 2. It consists in (1) developing a PNS tree until a given threshold and (2) regularly sending partial results to the server.

3 BREAKTHROUGH and Race Patterns

In this section we discuss the rules of BREAKTHROUGH (3.1), the retrograde analysis for small boards (3.2), and the race patterns (3.3).

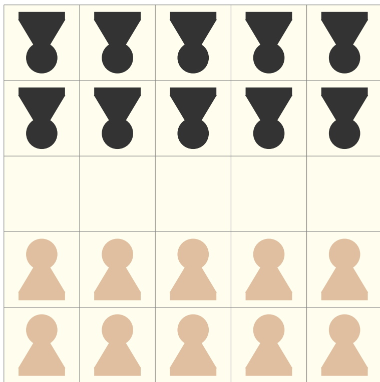
3.1 Rules of BREAKTHROUGH

BREAKTHROUGH is race game invented in 2001 by Dan Troyka. The game is played on a rectangular board of size 8×8. Each player starts with two rows of

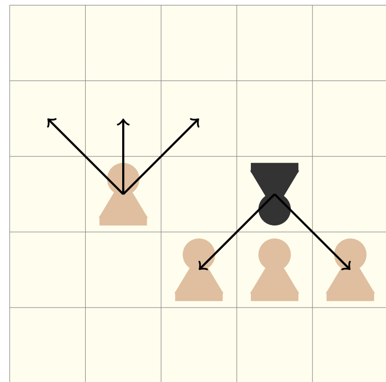
Algorithm 2. Remote algorithm.

```
while the stop message is not received do
  receive a position, a player and a threshold N from the server
  while root is not proved and number of descents is less than N do
    if the number of descent is a multiple of a parameter p then
      send as partial results to the server the current PN and the DN of the root
    end if
    expand the tree using Proof Number Search
  end while
  send the definitive results to the server
end while
```

pawns situated on opposite borders as shown in Figure 1(a). The pawns progress in opposite direction and the first player to bring a pawn to the opposite last row wins the game. A pawn can (1) always move diagonally forward possibly capturing an opponent pawn and (2) move forward one cell only if the cell is empty (Figure 1(b)).



(a) Starting position on size 5×5.



(b) Possible movements.

Fig. 1. Rules for the game BREAKTHROUGH

BREAKTHROUGH was originally designed to be played on a 7×7 board but was adapted to participate in the 8×8 board game-design competition which it won [10].

3.2 Retrograde Analysis for Small Boards

The state space complexity of BREAKTHROUGH on a board $m \times n$ with $m \geq 2$ and $n \geq 4$ can be upper bounded by the following formula $2^{2m} \times 3^{(n-2)m}$. This formula derives from the fact that each cell on the top row can only be empty or Black, each cell on the bottom row can only be empty or White, and all three

possibilities are available for cells in the $n-2$ central rows. This upper bound is relatively accurate for boards with a small height, but it includes positions that cannot be reached from the standard starting position as it does not take into account the fact that each player has at most $2m$ pieces. As a result, the upper bound is rather loose for larger boards.

Jan Haugland used retrograde analysis to solve BREAKTHROUGH on small boards.[2] The largest sizes solved by his program were 5×5 and 3×7, both turned out to be a second-player win.

To reduce the state space complexity and ease the retrograde analysis, Haugland avoided to store positions that could be won in one move. That is, positions with a pawn on the one but last row were not stored. It allows to reduce the state space complexity to $2^{4m} \times 3^{(n-4)m}$. The reduction factor is $r = \frac{3}{2}^{2m}$ which is $r = 58$ when $m = 5$.

3.3 Race Patterns

After a couple of games played, human players start to obtain some feeling for tactics in BREAKTHROUGH. It allows the experimented player to spot a winning path sometimes as early as 15 plies before the actual game end. A game of BREAKTHROUGH proceeds as follows, in the opening, the players strive to control the center or to obtain a strong outpost on the opponent's side without exchanging many pieces. Then, the players perform waiting moves until one of them enters a zugzwang position and need to weaken his[3] structure. The opponent will now try to take advantage of the breach, usually the attack involves sacrificing one or two pawns to force the opponent's defense to collapse. Thus, at this point both players could *break through* if the opponent passed, and the paths of both are usually disjoint, therefore it is necessary to count the number of moves needed by both players and the quickest to arrive wins (Figure 3(a) is an example of such a situation).

As we can see, detecting an early win involves looking separately at the possible winning paths of both players and deciding which is the shortest. Formalizing this technique can improve the playing level of an artificial player or the performance of a solver.

Defining Race Patterns. We define *race patterns* that allow to spot such winning paths. To be able to deal softly with the left and right sides of the board, we will consider a generalization of BREAKTHROUGH with walls.[4] Walls are static cells which can neither be traversed nor occupied by any player.

In the following, we assume that we are looking for a winning path for player White. Formally, a *pattern* for player White is a two dimensional matrix in which each element is of one of the following type {*occupied*, *free*, *passive*, *crossable*, *don't care*}. The representation and the relationship between these types is presented in Figure 2. A cell of type *passive* should not contain a black pawn to

[2] Available on http://www.neutreeko.net/neutreeko.htm.

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

[4] This generalization was used in the 2011 GGP competition.

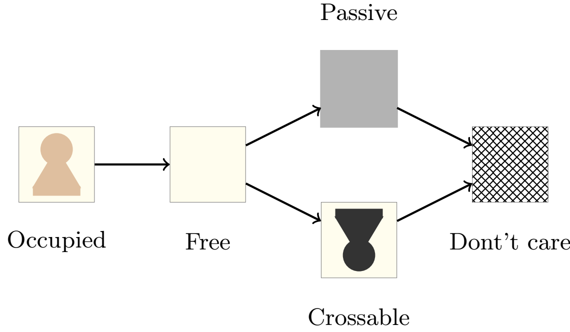


Fig. 2. Pattern representation. An arrow from a to b indicates that any cell satisfying a satisfies b.

Table 2. Checking race patterns for White

	Occupied	Free	Passive	Crossable	Don't Care
White Pawn	✓	✓	✓	✓	✓
Empty cell		✓	✓	✓	✓
Black Pawn				✓	✓
Wall			✓		✓

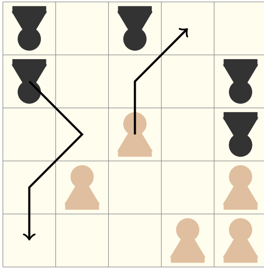
begin with, but it will not be necessary for any white pawn to cross it. On the other hand it should be allowed to bring a white pawn on a cell of type *crossable*, so it cannot be a wall but it could already hold a white or a black pawn.

To verify whether a pattern is matched on a given board, we first extend the board borders with walls and then check for each possible pattern location that every cell is compatible as defined by Table 2. For instance, if the attacking player is White, then a white pawn will match any cell type in the pattern. Stated otherwise, if the pattern cell corresponding to a black pawn is not *Crossable* or *Don't Care*, then the pattern does not match.

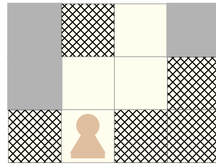
The *order* of a race pattern is defined as the maximal number of pass moves that Black is allowed to do before White wins in the restricted position designated by the race patterns.

We compute for each player the lowest-order matching race pattern and if they only intersect on *don't care* cells, we know the outcome of the game. For instance in Figure 3(a), we can see that White has two-move second-player win pattern (Figure 3(b)) and that Black has a three-move first-player win pattern (Figure 3(c)). Given that player Black does neither have a one-move nor a two-move race pattern, we may conclude that the position is a white win. It is thus possible to statically solve this position four moves before the actual game end.

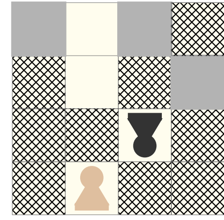
In general, this technique allows to solve positions $2 \times n$ moves before the actual game end, only if we have access to every n-move race pattern. However, a position cannot be solved this way if its solution tree involves a zugzwang.



(a) Sample game with Black to play. White player can force a win.



(b) Two-move second player win pattern.



(c) Three-move first player win pattern.

Fig. 3. Early win detection using race patterns. White can match pattern (b) and Black can match pattern (c)

In our experiments, we used 26 handwritten patterns of order up to 2. The biggest patterns we used were 4×3 such as the one presented in Figure 3(b). We do not have yet a tool for automatic correctness checking, therefore we had to limit the number of patterns used to keep confidence in their correctness.

4 Experimental Results

Experiments are done on a network of Linux computers connected with Gigabyte switches. The network area includes 17 computers with 3.2 GHz Intel i5 quad core CPU with 4 GB of RAM. The master is run alone on one of these computers. The maximum number of clients is set to 16×4.

In the following experiments, we report the total time needed to solve the starting position of a BREAKTHROUGH game of various sizes. We also report the number of nodes expanded and touched that were needed in Algorithm 1.

A node is expanded when all of its children have been added to the tree. In our server-side implementation, one node is expanded per iteration. The number of nodes expanded is proportional to the memory needed to store the PN tree on the server side. It also corresponds to the total number of tasks that have been sent to the clients. For a touched node, we only store the proof and disproof numbers as given by a client search. In contrast, an expanded node also needs to store a pointer to every child. As a result touched nodes take much less memory than expanded nodes. We bounded the number of descents in one search in the clients to 1k or 100k, so memory resources in the clients were never a problem in these experiments.

4.1 Scalability

Table 3 gives the time needed in seconds, the number of expanded and touched nodes saved on the server side to solve the 4×5 game with the PPN^2 algorithm

Table 3. Time needed, number of expanded and touched nodes for the PPN^2 with fixed search size on 4×5 board with 1k descents at most in the remote search. Partial results were sent from a client every 100 descents.

Clients	Time	Speed-up	Expanded	Touched
1	3397s		107k	915k
4	1559s	2.2	126k	1073k
8	803s	4.2	130k	1106k
16	472s	7.2	152k	1298k
32	305s	11.1	196k	1651k
64	186s	18.3	232k	1930k

with 1k descents in the clients. Solving 4×5 with 64 clients with 100k descents in each remote search takes 577 seconds, while with 10k descents in each remote search, it takes 216 seconds. Therefore, increasing the number of descents in the remote search does not necessarily improve the solving time. In contrast, performing 100 descents in each remote search made it necessary to go over 1m descents in the main search which is quite memory consuming.

From Table 3 we can see , that the number of expanded nodes on the server increases as the number of clients rises. Stated otherwise, running many clients in parallel makes it harder to avoid unnecessary work. This is an expected behavior in a parallel algorithm. Nevertheless, the time needed to solve the position also decreases steadily as the number of clients rises. The speedup factor with 8 and 64 clients compared to 1 single client are respectively 4 and 18. So, we may conclude that although the algorithm is not perfectly parallelizable, the scaling factor is satisfactory.

4.2 Partial Results Updates

Table 4 gives the time needed in seconds, the number of expanded and touched nodes saved in memory to solve the 4×5 game with the PPN^2 algorithm with partial results. The first column gives the partial results frequency. Each solved position turned to be a second-player win.

As we can see, sending partial results makes it possible to direct better the search but also increases the communication overhead. It is therefore needed to find a balance between spending too much time in communications and not taking advantage of the information available. In this setting, sending partial results every 100 descents in the client seems the best compromise. When using partial informations, the solving time is less dependent to the search size.

4.3 Patterns

Table 5 gives the time in seconds and the number of expanded nodes needed to solve different games with the PPN^2 algorithm with partial results, fixed

Table 4. Time needed, number of expanded and touched nodes for PPN^2 algorithm with partial results and fixed remote search size of 1k descents on 4×5 board, involving 64 clients.

Partial	Time	Expanded	Touched
None	263s	324k	2645k
500	233s	281k	2336k
250	205s	253k	2105k
100	186s	232k	1930k
50	190s	233k	1944k
25	201s	243k	2023k
12	193s	223k	1855k

search size and some patterns. The patterns we used allowed to solve statically a position up to 4 moves before the game end, 26 patterns were hand-written for this purpose. Checking whether a pattern can be matched on a given position is done in the most naive way and implementing more efficient pattern matching techniques is left as future work.

Using race patterns, the solving time is divided by 5.96 for the 4×5 board with 1k search, and by 9.85 for the 5×5 board. It takes 927 seconds to solve the 5×5 board with 1k search in the clients. Without patterns, 5×5 board with 1k search fails with 1 million nodes saved and goes beyond the server allowable memory with 2 million nodes.

Combining PPN^2 and race patterns allows us to solve the 6×5 board in 25,638 seconds (i.e., 7 hours 7 minutes 18 seconds) with 10k search and in 47,134 seconds (i.e., 13 hours 5 minutes 34 seconds) with 100k search.

As we can see, using race patterns makes it unnecessary to examine many positions in the main search. Race patterns also allow for a time reduction of one order of magnitude on boards of small sizes and probably more on larger boards.

Table 5. Time needed and number of expanded nodes for the PPN^2 algorithm with partial results, fixed remote search size and patterns with 64 clients

Board size	Search size	Patterns	Time	Expanded
5×4	1k	No	2s	4132
5×4	1k	Yes	1s	72
4×5	1k	No	161s	241k
4×5	1k	Yes	27s	4k
5×5	1k	Yes	927s	78k
5×5	100k	No	29,170s	208k
5×5	100k	Yes	2959s	3k
6×5	10k	Yes	25,638s	14k
6×5	100k	Yes	47,134s	21k

5 Discussion and Conclusion

In this paper, we have defined race patterns and used them to ease the solving of BREAKTHROUGH positions. Indeed, in our experiments, using race patterns typically allows to examine about two orders of magnitude fewer positions. We have also shown how to parallelize successfully the PN^2 algorithm. The PPN^2 algorithm associated to race patterns has enabled to solve 6×5 BREAKTHROUGH: the game is a second-player win. We have found that on the smaller 4×5 board the speedup due to parallelization is important until at least 64 clients.

In future work, we will try to solve BREAKTHROUGH for larger sizes. The race patterns used in this work had been devised by hand, but it is impractical if we need many more patterns to solve statically positions earlier. We therefore need to devise an algorithm to generate the race patterns and check them for correctness automatically.

Zugzwang positions are still difficult to solve. Indeed, no winning race pattern will be found in a zugzwang position, so an extension of the concept of race patterns to be compatible with zugzwang positions or an orthogonal technique would be desirable.

We will also apply the Parallel PN^2 algorithm to other games. Moreover we will try to enhance the algorithm itself in order to have even greater speedups.

References

1. Allis, L.V.: Searching for Solutions in Games an Artificial Intelligence. Phd thesis, Vrije Universitat Amsterdam, Department of Computer Science, Rijksuniversiteit Limburg (1994)
2. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-Number Search. Artificial Intelligence 66(1), 91–124 (1994)
3. Breuker, D.M.: Memory versus Search in Games. Phd thesis, Universiteit Maastricht (1998)
4. Brockington, M.: Asynchronous Parallel Game-Tree Search. Phd thesis, University of Alberta (1997)
5. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Proceedings of the Computer Games Workshop, pp. 93–101 (2007)
6. 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)
7. Enzenberger, M., Müller, M.: A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 14–20. Springer, Heidelberg (2010)
8. Feldmann, R., Mysliwietz, P., Monien, B.: Game tree search on a massively parallel system, In: Advances in Computer Chess 7, pp. 203–219 (1993)
9. Finnsson, H., Björnsson, Y.: Game-tree properties and mcts performance. In: Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA 2011), pp. 23–30 (2011)
10. Handscomb, K.: 8×8 game design competition: The winning game: Breakthrough... and two other favorites. Abstract Games Magazine 7, 8–9 (2001)

11. Kaneko, T.: Parallel depth first proof number search. In: AAAI (2010)
12. Kishimoto, A., Schaeffer, J.: Distributed game-tree search using transposition table driven work scheduling. In: Proceedings International Conference on Parallel Processing, pp. 323–330. IEEE (2002)
13. Kishimoto, A., Kotani, Y.: Parallel and/or tree search based on proof and disproof numbers. In: Game Programming Workshop in Japan, pp. 24–30 (1999)
14. Nagai, A.: A new and/or tree search algorithm using proof number and disproof number. In: Complex Games Lab Workshop, pp. 40–45. ETL, Tsukuba (1998)
15. Nagai, A.: Df-pn algorithm for searching AND/OR trees and its applications. Ph.D. thesis (2002)
16. Saito, J.-T., Winands, M.H.M., van den Herik, H.J.: Randomized Parallel Proof-Number Search. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 75–87. Springer, Heidelberg (2010)
17. Schadd, M.P.D., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bergsma, M.H.J.: Best Play in Fanorona leads to Draw. New Mathematics and Natural Computation 4(3), 369–387 (2008)
18. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., 0003, M.M., Lake, R., Lu, P., Sutphen, S.: Solving checkers. In: IJCAI, pp. 292–297 (2005)
19. Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science 317(5844), 1518 (2007)
20. Skowronski, P., Björnsson, Y., Winands, M.: Automated Discovery of Search-Extension Features. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 182–194. Springer, Heidelberg (2010)
21. Wu, I.-C., Lin, H.-H., Lin, P.-H., Sun, D.-J., Chan, Y.-C., Chen, B.-T.: Job-Level Proof-Number Search for Connect6. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 11–22. Springer, Heidelberg (2011)

Infinite Connect-Four Is Solved: Draw

Yoshiaki Yamaguchi, Kazunori Yamaguchi,
Tetsuro Tanaka, and Tomoyuki Kaneko

The University of Tokyo
yoshiaki@graco.c.u-tokyo.ac.jp

Abstract. In this paper, we present the newly obtained solution for variants of Connect-Four played on an infinite board. We proved this result by introducing never-losing strategies for both players. The strategies consist of a combination of paving patterns, which are follow-up, follow-in-CUP, and a few others. By employing the strategies, both players can block their opponents to achieve the winning condition. This means that optimal play by both players leads to a draw in these games.

By rearrangement of the same paving patterns, the solution for a semi-infinite board, where either the height or the width is finite, are also presented. Moreover, it is confirmed that these results are effective under various placement restrictions.

1 Introduction

The solution of a game has been one of the main targets of game research [1]. In this paper, we introduce the newly obtained solution for variants of Connect-Four played on a board infinite in height, width, or both. We prove this result by introducing never-losing strategies for both players. By employing the strategies, both players can block the opponents to achieve the winning condition. This means that optimal play by both players leads to a draw in these games. The strategies for each variation consist of a combination of common paving patterns. We also confirm that the result is effective under placement restrictions.

First, we introduce the rule of Infinite Connect-Four (the variant of Connect-Four played on a board infinite in both height and width).

- Two players, denoted by X (first player, White) which plays first and O (second player, Black) which plays second, in turn place a disk in a cell on an infinite board. A *cell* is specified by row and column numbers. A row number is a positive integer, and a column number is an integer: positive, negative or zero.
- Each player places his[1] disk denoted by the same symbol of the player: X or O.
- A disk can be placed at the lowest unoccupied cell for each column.
- By "X moves i" we mean that X places a disk in a column i.
- Initially, X moves 0.

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

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 208–219, 2012.
© Springer-Verlag Berlin Heidelberg 2012

– If a player obtains a group of four connected disks of his symbol either horizontally, vertically, or diagonally, he wins the game. The arrangement of disks is called *Connect4*.
– If either player wins the game, the game is over.
– If neither X nor O achieves Connect4, the game will never end and is drawn.

We call a configuration of disks and turn a *position*. A disk which is already placed in a position is denoted in bold roman as **X** or **O** and a disk to be placed in a position is denoted in italic as *X* or *O*.

An example position is presented in Fig. 1. We present a position so that a column number increases from left to right and a row number increases upward as presented in Fig. 1.

A position can be specified by moves from the empty board to the position. Because a disk is placed at the row of the lowest number for each column, a move can be specified only by a column number. Thus we use a sequence of column numbers preceded by player symbols to specify a position. For example, we denote a position after X moved 0, O moved -1, X moved -1, O moved 1, X moved 1, and O moved 2 by X0O-1X-1O1XO2 (Fig. 1).

2 Previous Work

In 1988, it was proved by James Dow Allen [2] that in Connect-Four X wins; independently and almost simultaneously it was proved, too, by Victor Allis [3]. Both authors applied a different method. The results of the game played on finite boards of some non-standard heights and widths were reported in [7]. Table 1 shows the result.

$Connect(m, n, k, p, q)$ was proposed in [6]. $Connect(m, n, k, p, q)$ is a game played by X and O on the $m \times n$ board by placing p disks in each turn at a time with q disks placed initially to make a k-in-a-row. In [5], $Connect(\infty, \infty, k, p, p)$ was analyzed and it was presented that $Connect(\infty, \infty, 11, 2, 2)$ and $Connect$ $(\infty, \infty, 3p + 3d(p) + 8, p, p)$ for $p \geq 3$ for a logarithmic function $d(p)$ are drawn (the game never ends). The method used is to decompose the board into a combination of sub-boards and use the property that p-in-a-row exists if $3p$-in-a-row is in a sub-board. The case for $p = 1$ is not handled in the paper. Apparently, the case was left for future work. Also, $Connect(m, n, k, p, q)$ does not include Connect-Four and Infinite Connect-Four because Connect-Four and Infinite Connect-Four have "ground" under the 1st row and "gravity" to attract disks to lower rows.

3 Solution of Infinite Connect-Four

The result of a two-person zero-sum game of perfect information is called a *draw* when both players cannot win for some opponent's moves from the starting position.

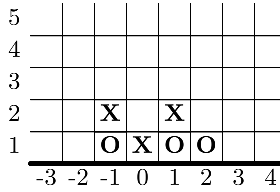


Fig. 1. An Example Position of Infinite Connect-Four

Table 1. Game-Theoretic Value of Connect-Four on Finite Boards [7]

height									
11	=								
10	=	=							
9	=	=	X						
8	=	=	O	X					
7	=	=	X	=	X				
6	=	=	O	X	O	O			
5	=	=	=	=	X	X	X		
4	=	=	O	=	O	O	O	O	
	4	5	6	7	8	9	10	11	width

Theorem 1. *Infinite Connect-Four is solved: Optimal play by both players leads to a draw.*

We prove Theorem 1 by presenting strategies for both players to play Infinite Connect-Four without losing a game.

First, we introduce some definitions.

Definition 1. ***(Initial Cell)*** *We define the initial cell as the cell for X's first move. The initial cell is placed in the 1st row and 0th column of a board.*

In this paper, we employ strategies such as the paving strategy in the Polyomino Achievement Game [8].

Definition 2. ***(Tile)*** *A pair of two cells used to block both cells to be occupied by the opponent's disks is called a tile.*

A tile can be a pair of horizontally or vertically adjacent cells, but sometimes it can be a pair of non-adjacent cells. Fig. 2 shows an example of a horizontal tile. Assume that this tile is included in the never-losing proof for O. In this case, if X moves 0, O must move 1 immediately to prevent X from occupying the both cells in the tile.

Definition 3. ***(Follow-in-tile; Follow-up)*** *Placing one's disk in a tile after an opponent placed a disk in the tile is called follow-in-tile. Especially, placing one's disk in a vertical tile after an opponent placed a disk in the tile is called follow-up[3].*

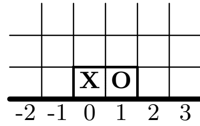


Fig. 2. Tile

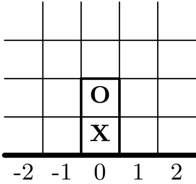


Fig. 3. Follow-up in Vertical Tile

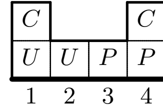


Fig. 4. CUP

When a player plays follow-up in a tile, the vertical order of two disks in the tile is uniquely determined such that the player's disk is on the opponent's disk because of gravity. Fig. 3 shows the position after O played follow-up in a vertical tile.

Definition 4. ***(CUP)*** *A CUP is a CUP-shaped combination of three tiles as illustrated in Fig. 4. We denote each tile in a CUP by C, U and P. We only consider CUPs placed on the ground in our proof.*

In some figures such as Fig. 4 and some explanations in this paper, we number the columns relatively with the leftmost column number set to 1 for conciseness.

Lemma 1. *For each CUP, if X and O place a disk in turn with X at the first move, there is a strategy for O to force X to fill at most one of the C, U, and P cells. Similarly, if X and O place a disk in turn with O at the first move, there is a strategy for X to force O to fill at most one of the C, U, and P cells.*

Proof. We present a strategy for O and show that it satisfies the condition by case analysis. We can assume that X moves either of U because of the symmetry. For X1 (Fig. 5), O moves the other U (Fig. 6). Note that the column numbers in this proof are relative ones. Then, X can move either of P (Fig. 7) or C in the 1st column (Fig. 11). For X1O2X3, O moves the other P (Fig. 8). Then, X may move either of C (Fig. 9). O moves the other *C* (Fig. 10). In Fig. 10, O forced X to fill at most one *C*, *U*, and *P* cells. Similarly, O can force X to fill at most one *C*, *U*, and *P* cells in the cases for X1O2X1 or X2O1X1 (Fig. 11). Because O cannot move *C* in the 4th column, O cannot play follow-in-tile immediately in these cases. O moves *P* in the 3rd column (Fig. 12). Because also X cannot

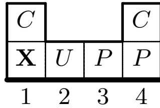


Fig. 5. CUP after X1

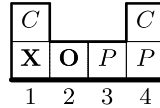


Fig. 6. CUP after X1O2

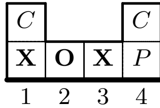


Fig. 7. CUP after X1O2X3

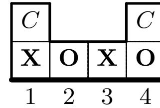


Fig. 8. CUP after X1O2X3O4

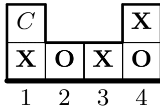


Fig. 9. CUP after X1O2X3O4X4

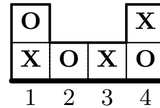


Fig. 10. CUP after X1O2X3O4X4O1

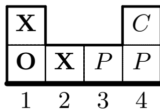


Fig. 11. CUP after X2O1X1

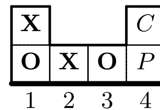


Fig. 12. CUP after X2O1X1O3

move C in the 4th column, X is forced to move the P (Fig. 13) if X tries to move C in the 4th column next. O moves C in the 4th column (Fig. 14) and even in this case, O forced X to fill at most one C, U, and P cells. A strategy for X to force O to fill at most one C, U, and P cells is similar.

Definition 5. ***(Follow-in -CUP)*** *Follow-in-CUP is a strategy for a player to prevent the opponent from occupying the both cells in any of C, U or P tile.*

Definition 6. ***(X-Board)*** *X-Board is a strategy for X consisting of the initial cell, CUPs and vertical tiles as illustrated in Fig. 15.*

- *The initial cell is at the 1st row and the 0th column, by definition.*
- *CUPs are placed in the columns* $5n+3\ldots5n+6$ *(inclusive) and* $-5n-6\ldots-5n-3$ *(inclusive) for* $n \in N_0$ *(*$N_0 = N \cup \{0\}$*).*
- *Other cells are filled with vertical tiles.*

If O moves in a CUP, X plays follow-in-CUP in the CUP. If O moves in a vertical tile, X plays follow-up in the tile.

If X moves along the X-Board, the layout of disks is determined uniquely except that in a tile in CUPs.

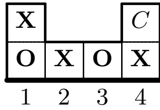


Fig. 13. CUP after X2O1X1O3X4

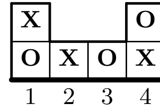


Fig. 14. CUP after X2O1X1O3X4O4

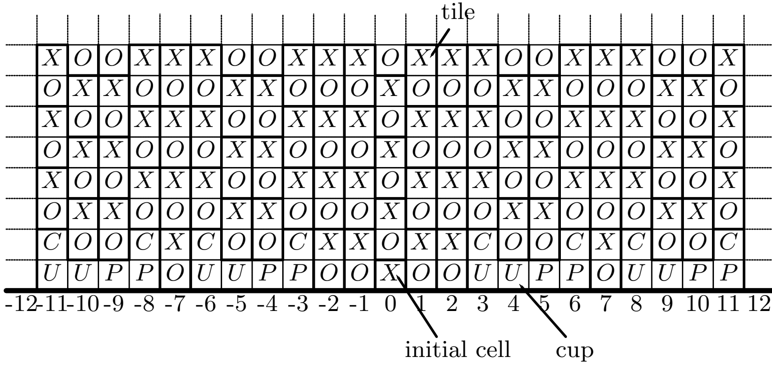


Fig. 15. X-Board

Lemma 2. *If X moves along X-Board, X never loses.*

Proof. If X moves along X-Board and Connect4 is made by X or O, then Connect4 of X or O should be in X-Board because no disk is removed or replaced in the course of a game. In order to present that there is no Connect4 in X-Board, we analyze the sub-board from the −1st to the 9th columns and from the 1st to the 7th rows (name it *Core77*) as presented in Fig. 16.

In X-Board, the layout of CUPs and vertical tiles in a sub-board in $5n + 7 \ldots 5n + 11$ (inclusive) columns $n \in N_0$ ($N_0 = N \cup \{0\}$) is identical to that in the 2nd to the 6th columns. The layout of CUPs and vertical tiles in a sub-board in $-5n - 11 \ldots -5n - 7$ (inclusive) columns $n \in N_0$ ($N_0 = N \cup \{0\}$) is identical

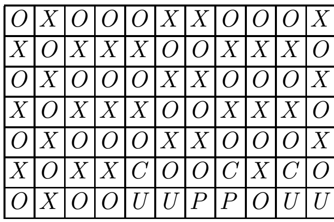


Fig. 16. Core77

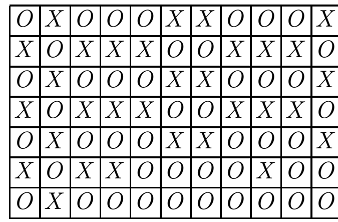


Fig. 17. Core77 with all C, U, and P replaced with O

O	X	O	O	O	X	X	O	O	O	X
X	O	X	X	X	O	O	X	X	X	O
O	X	O	O	O	X	X	O	O	O	X
X	O	X	X	X	O	O	X	X	X	O
O	X	O	O	O	X	X	O	O	O	X
X	O	X	X	X	O	O	X	X	X	O
O	X	O	O	X	X	X	X	O	X	X

Fig. 18. Core77 with all C, U, and P replaced with X

to that in the -6th to the -2nd columns. Thus, any layout of CUPs and vertical tiles in a 4×4 sub-board in X-Board appears in some 4×4 sub-board in Core77 except vertical (left-right) symmetry. So proving that there is no Connect4 in Core77 is sufficient for proving there is no Connect4 in X-Board.

From Core77 with all C, U, and P replaced with O (Fig. 17) or X (Fig. 18), we can readily see that there is no vertical or diagonal Connect4. It is easy to see that there is no horizontal Connect4 from the 3rd row to the 7th row in Core77. Because each of C, U, and P in a CUP cannot be occupied by disks of the same player, there is no Connect4 in the 1st and the 2nd rows of Core77. This completes the proof that there is no Connect4 in Core77, no Connect4 in X-Board, X and O cannot make Connect4, and the game is drawn if X moves along X-Board.

Note that X-Board is not an optimal strategy for X if O does not take an optimal strategy for O. For example, from the position in Fig 1, X can make Connect4 by X0O2X0O0X-2O-2X1O1X-1O2, but this deviates from X-Board.

Next, we consider a strategy for O. Similarly to X-Board, we define *O-Board* as presented in Fig. 19.

Definition 7. ***(O-Board)*** *We call the following strategy O-Board.*

- *CUPs are placed in the columns* $5n+1 \ldots 5n+4$ *(inclusive) and* $-5n-4 \ldots -5n-1$ *(inclusive) for* $n \in N_0$ *(*$N_0 = N \cup \{0\}$*).*
- *Other cells are filled with vertical tiles.*

If X places a disk in a CUP, O plays follow-in-CUP in the CUP. If X places a disk in a tile, O plays follow-up in the tile.

If O moves along the O-Board, the layout of disks is determined uniquely except that in a tile in CUPs.

Lemma 3. *If O moves along O-Board, O never loses.*

Proof. If O moves along O-Board, disks are placed according to O-Board, and it can be proved that there is no Connect4 using Core77 with reversed X and O as before except symmetry.

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	
	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	
	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	
	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	
	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	O	X	X	O	O	
	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	X	O	O	X	X	
	O	C	X	X	C	O	C	X	X	C	O	C	X	X	C	O	C	X	X	C	O	
	X	U	U	P	P	X	U	U	P	P	X	U	U	P	P	X	U	U	P	P	X	

Fig. 19. O-Board

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
											O											
											X											
											O											
											X											
											O											
											X											
											O											
											X											

Fig. 20. Result When X Moves along X-Board and O Moves along O-Board

Proof (Proof of Theorem 1). By Lemmas 2 and 3, X and O can achieve at least a draw against any opponent. Therefore Infinite Connect-Four is solved: Optimal play by both players leads to a draw.

If X moves along X-Board and O moves along O-Board, they play only follow-up infinitely after the initial move (Fig. 20).

4 Solution of Variants of Infinite Connect-Four

4.1 Placing Rules

In the previous section, players can place a disk in any column. We call this rule *Free Placement Rule*. There are the following local rules in Infinite Connect-Four forbidding players to place a disk far from the already placed disks [4].

- *Next Placement Rule*: A player can place a disk in a column i if there is at least one disk in the columns $i-1 \dots i+1$ (inclusive) or $i = 0$. As in the case of Fig. 1, X can place a disk from the -2nd column to the 3rd column.

- *Space Placement Rule*: A player can place a disk in a column i if there is at least one disk in the columns $i-2\dots i+2$ (inclusive) or $i=0$. As in the case of Fig. 1, X can place a disk from the -3rd column to the 4th column.
- *Free Placement Rule*: A player can place a disk in any column.

Theorem 2. *Infinite Connect-Four under any of the three placement rules is solved: Optimal play by both players leads to a draw.*

Proof. Even if X plays under Next Placement Rule and O plays under Free Placement Rule, X can move along X-Board and achieve at least a draw against any opponent. So, X can achieve at least a draw under any placement rule. So can O.

4.2 Height Limit

In this section, we present the solution for such cases that the height (the number of rows) is limited while the width (the number of columns) is unlimited. Here we consider only Free Placement Rule.

Theorem 3. *Infinite Connect-Four in any height limited board under Free Placement Rule is solved: Optimal play by both players leads to a draw.*

We prove Theorem 3 by presenting strategies for both players to play Infinite Connect-Four in any height limited board without losing the game.

Height 1. X plays follow-in-tile for horizontal tiles of $\{n, n+1\}$ for $n = \{1, 3, 5, \cdots\}$ and tiles of $\{n, n-1\}$ for $n = \{-1, -3, -5, \cdots\}$ except the initial 0. This prevents O from making even a three-in-a-row. O plays follow-in-tile for horizontal tiles of $\{n, n+1\}$ for $n = \{\cdots -4, -2, 0, 2, 4, \cdots\}$. This prevents X from making even a three-in-a-row.

Height More than 1. We employ X-Board truncated to the specified height h. In the truncated X-Board, tiles spanning in the rows $h...h+1$ are chopped. If O places a disk in one of the chopped tiles, X cannot play follow-up. In such a case, we adjust the truncated X-Board so that X can safely place his disk without disrupting tiles in the original strategy. To make the insertion harmless to the already placed disks, X finds the CUP (name it xCUP) just right to the rightmost disk. We insert four columns just right to xCUP. The first, second, and fourth of the columns consist of just tiles and the third of the column consists of a cell on the ground and tiles on it. After the insertion, the columns in Fig. 21 are changed to the ones shown in Fig. 22, and X moves to the third column of the inserted columns. The layout of CUPs and vertical tiles in these columns are already included in Core77 except symmetry and the non-existence of Connect4 after the insertion is easily confirmed.

Definition 8. ***(Height-limited-X-Board)*** *Height-limited-X-Board is a strategy determined by the truncated X-Board and its variants with inserted columns.*

C	*O*	*O*	*C*	*X*	*C*	*O*	*O*	*C*
U	*U*	*P*	*P*	*O*	*U*	*U*	*P*	*P*

Fig. 21. xCUP and the One Next Right

C	*O*	*O*	*C*	*X*	*X*	*O*	*X*	*X*	*C*	*O*	*O*	*C*
U	*U*	*P*	*P*	*O*	*O*	**X**	*O*	*O*	*U*	*U*	*P*	*P*

Fig. 22. Insertion of Four Columns between Columns in Fig. 21

···	0	1	2	3	4	5	6	7	8	9	10	11	12	···
	X	**O**	*O*	*O*	*X*	*X*	*O*	*O*	*O*	*X*	*X*	*O*	*O*	
···	*O*	**X**	*X*	*C*	*O*	*O*	*C*	*X*	*C*	*O*	*O*	*C*	*X*	···
	X	**O**	*O*	*U*	*U*	*P*	*P*	*O*	*U*	*U*	*P*	*P*	*O*	

Fig. 23. Follow-Up in 1st column Failed by Height Limit

Lemma 4. *If X moves along height-limited-X-Board, X never loses.*

Proof. Any layout of CUPs and vertical tiles in a 4×4 sub-board in height-limited-X-Board appears in some 4×4 sub-board in Core77 except vertical (left-right) symmetry.

We present an example for height 3. After X moves 0 initially, O moves 1, X moves 1, and O moves 1, follow-up in the column 1 is impossible in Fig. 23. At this point, X inserts four columns between the columns 6 and 7 and X moves 9 in the resulting height-limited-X-Board.

Next, we present the strategy to achieve a draw for O. O moves along the O-Board truncated to the specified height. If O cannot play follow-in-tile due to height limit, then O finds the CUP (name it oCUP) just right to the rightmost disk. Then, O inserts the four columns of the Xs insertion with reversed X and O just right to oCUP and moves to the third column of the inserted columns. This changes the columns in Fig. 25 to the ones shown in Fig. 26.

Definition 9. ***(Height-limited-O-Board)*** *Height-limited-O-Board is a strategy determined by the truncated O-Board and its variants with inserted columns.*

Lemma 5. *If O moves along height-limited-O-Board, O never loses.*

Proof. Any layout of CUPs and vertical tiles in a 4×4 sub-board in height-limited-O-Board appears in some 4×4 sub-board in Core77 except vertical (left-right) symmetry.

···	0	1	2	3	4	5	6	7	8	9	10	11	12	···
	X	**O**	*O*	*O*	*O*	*X*	*X*	*O*	*O*	*X*	*O*	*O*	*O*	
···	*O*	**X**	*X*	*C*	*O*	*O*	*C*	*X*	*X*	*O*	*X*	*X*	*C*	···
	X	**O**	*O*	*U*	*U*	*P*	*P*	*O*	*O*	**X**	*O*	*O*	*U*	

Fig. 24. A Height-limited-X-Board

C	*X*	*X*	*C*	*O*	*C*	*X*	*X*	*C*
U	*U*	*P*	*P*	*X*	*U*	*U*	*P*	*P*

Fig. 25. oCUP and the One Next Right

C	*X*	*X*	*C*	*O*	*O*	*X*	*O*	*O*	*C*	*X*	*X*	*C*
U	*U*	*P*	*P*	*X*	*X*	**O**	*X*	*X*	*U*	*U*	*P*	*P*

Fig. 26. Insertion of Four Columns between Columns in Fig. 25

Proof (Proof of Theorem 3). By Lemmas 4 and 5, X and O can achieve at least a draw against any opponent. Therefore Infinite Connect-Four in any height limited board under any of the three placement rules is solved: Optimal play by both players leads to a draw.

4.3 Width Limit

In this section, we present the case that the width (the number of columns) is limited while the height (the number of rows) is unlimited. The leftmost column is numbered 0 in the following boards. We employ a modified rule that X can place a disk in any column initially for the game on the board with limited width.

Theorem 4. *Infinite Connect-Four in any width limited board under any of the three placement rules is solved: Optimal play by both players leads to a draw.*

We prove Theorem 4 by discovering strategies for both players to play Infinite Connect-Four in any width limited board without losing game. This result is applicable to all three rules: Free Placement Rule, Next Placement Rule, and Space Placement Rule. We proved this by showing X-Board and O-Board for each width. However, because of the space limitation, we omit this proof. For the complete proof of this theorem, please refer to the longer version of this paper available at `http://www.graco.c.u-tokyo.ac.jp/~yoshiaki/hako-long.pdf`.

5 Conclusion

We proved that the game result of Infinite Connect-Four is drawn (the game never ends). The paving strategies employed to prove it is so general that with slight modifications it can be used to prove that the game result of Infinite Connect-Four on a vertically limited or horizontally limited board is also drawn.

References

1. van den Herik, H.J., Uiterwijk, J.W.H.M., van Rijswijck, J.: Games solved: now and in the future. Artificial Intelligence 134(1-2), 277–311 (2002)
2. Allen, J.D.: A Note on the Computer Solution of Connect-Four. In: Levy, D.N.L., Beal, D.F. (eds.) Heuristic Programming in Artificial Intelligence, The first Computer Olympiad, pp. 134–135. Ellis Horwood, Chichester (1989)

3. Allis, L.V.: A Knowledge-based Approach to Connect-Four, The Game is Solved: White wins: Master's thesis, Vrije Universiteit, Amsterdam (1988)
4. http://homepage3.nifty.com/yasda/dwnload/hako.htm (last accessed on February 3, 2011)
5. Chiang, S.-H., Wu, I.-C., Lin, P.-H.: On Drawn K-In-A-Row Games. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 158–169. Springer, Heidelberg (2010)
6. Wu, I.C., Huang, D.Y., Chang, H.C.: Connect6. ICGA Journal 28(4), 235–242 (2005)
7. Tromp, J.: Solving Connect-4 on Medium Board Sizes. ICGA Journal 31(2), 110–112 (2008)
8. Halupczok, I., Schlage-Puchta, J.C.: Achieving Snaky. Electronic Journal of Combinatorial Number Theory 7(1), 1–28 (2007)

Blunder Cost in Go and Hex

Henry Brausen, Ryan B. Hayward, Martin Müller,
Abdul Qadir, and David Spies

Computing Science, University of Alberta
{hbrausen,hayward,mmueller}@ualberta.ca

Abstract. In Go and Hex, we examine the effect of a blunder — here, a random move — at various stages of a game. For each fixed move number, we run a self-play tournament to determine the expected blunder cost at that point.

1 Introduction

To *blunder* — literally, to stumble blindly [9] — is to move stupidly, with apparent ignorance of one's situation. In the context of games, a blunder is usually defined as a serious mistake, for example a losing move when a winning move is available. But a mindless move need not be catastrophic. In the endgame, a blunder usually leads to a quick loss, but in opening play, a blunder might not even be noticed. So, what is the average cost of a *blunder* — namely, a random move — at various stages of a game?

Games that are interesting to play are typically hard to solve, so usually this question can be answered only experimentally rather than exactly. In this paper we present experimental blunder cost data for the Go player Fuego and the Hex player MoHex.

For a particular move number, blunder cost is a function of both player strength (the stronger the player, the higher the cost) and position difficulty (the greater the number of losing moves, the higher the cost). Thus *blunder analysis*, in which a player is compared with a blundering version of itself, can be useful as a diagnostic tool both for players and for games.

2 A Simple Model of Two-Player No-Draw Games

To explain how blunder analysis can be useful, we first outline a straightforward model of two-player no-draw games. (For other approaches of modeling more general games, see for example the work of Haworth and Regan [10,20].)

Our model's parameters are as follows:

- T: maximum number of moves in a game,
- t: moves made so far, in $\{0, \ldots, T\}$,
- e_t: computational ease of solving a state after t moves, from 0 (intractable) to 1 (trivial),

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 220–229, 2012.
© Springer-Verlag Berlin Heidelberg 2012

- w_t: fraction of available moves that are winning, from -1 (all losing) to 1 (all winning), with $w_t = -x < 0$ indicating that all player moves are losing, and furthermore that after the best move the opponent will have w_{t+1} close to x,
- m_t: score of move t, from -1 (weakest) to 1 (strongest),
- r_t: rank of move t, from 1 (weakest) to k_t (strongest), where k_t is the number of available moves,
- s_p: strength of player p, from -1 (anti-perfect, always likely to play the weakest move) to 0 (uniform random) to 1 (perfect, always likely to play the strongest move).

We assume that games are non-pathological, in the sense defined by Dana Nau [19], so that the smaller the search space, the greater the tendency to make a strong move. Thus e_t, which approximates the size of the search space, increases smoothly with t.

With these parameters, a game can be modeled as follows:

- for each move number t, compute m_t by sampling from a distribution with mean $s_p \times e_t$,
- then compute r_t from m_t,
- then make the move with rank r_t,
- then compute w_{t+1} from w_t and r_t, say by sampling from a distribution, with the sampling formula ensuring that the strongest move from a winning position always leaves the opponent with no winning moves.

So, in this model, what is the expected cost of a blunder? The probability of making a winning move depends on w_t, the fraction of winning moves available, and s_p, the player's strength. We assume that both players try to win and can perform some useful computation, and so $s_p > 0$ for both players. A blunder is a uniform random move: in our model, this corresponds to a move of which the strength is uniformly sampled from the interval [-1, 1]. Since non-blunder moves are sampled from a distribution D with expected value $s_p \times e_t$, a blunder is similar to (and, if D is uniform, identical to) a move made by a player whose strength temporarily drops to 0. Thus the expected cost of a blunder is the drop in win rate caused by this temporary strength loss. So, this form of blunder analysis gives an indirect measurement of $s_p \times e_t$, namely playing strength times ease of solving the game at move t.

In this paper, we consider what happens when the "blunder player" makes exactly one blunder per game. We also considered experiments where the blunder player makes two consecutive blunders, but this resulted in the blunder player's win rate being extremely low and difficult to measure, and so we do not include any of this data.

3 Blunder Analysis of Fuego

FUEGO is the open source Monte Carlo tree search Go program originally developed by the University of Alberta Computer Go group, led by Martin Müller

and including Markus Enzenberger, Broderick Arneson [17,8,16]. In 2009 FUEGO became the first computer Go program to win a 9×9 game without handicap against a top professional player, 9-dan Chou Chun-Hsun. FUEGO has won a number of computer Go tournaments, including the 2010 UEC Cup [15,7].

In our experiments, the default player never deliberately blunders. Baseline data, which shows the respective black (first player) and white (second player) win rates, is generated from a tournament between two default players. Each move-k win rate is an average of win rates from a trial between a blunder-player, who blunders only at that move, and a default player. The blunder-player data values are drawn as two curves, one for each possible blunder-player color. A third curve shows the fraction of games still active, namely unfinished, at that point. Each data value and each baseline value is computed from a 500 game trial.

Error bars show a binomial proportion confidence interval of $2\sqrt{(}p(1-p)/n)$, where p is the proportion and n is the number of trials, yielding a confidence of slightly more than 95%. Each move-k datum is computed only on games still active at move k, so error bars enlarge as the number of active games decreases.

In these Go experiments, White receives a komi of 7.5 points. Unless otherwise noted, FUEGO runs 10000 MCTS simulations per move, and the resign threshold is 0.05.

3.1 9×9 Go

Figure 1 shows the effect of FUEGO blunders on the 9×9 board. The baselines show black/white win rates of about 47%/53%, suggesting that on the 9×9

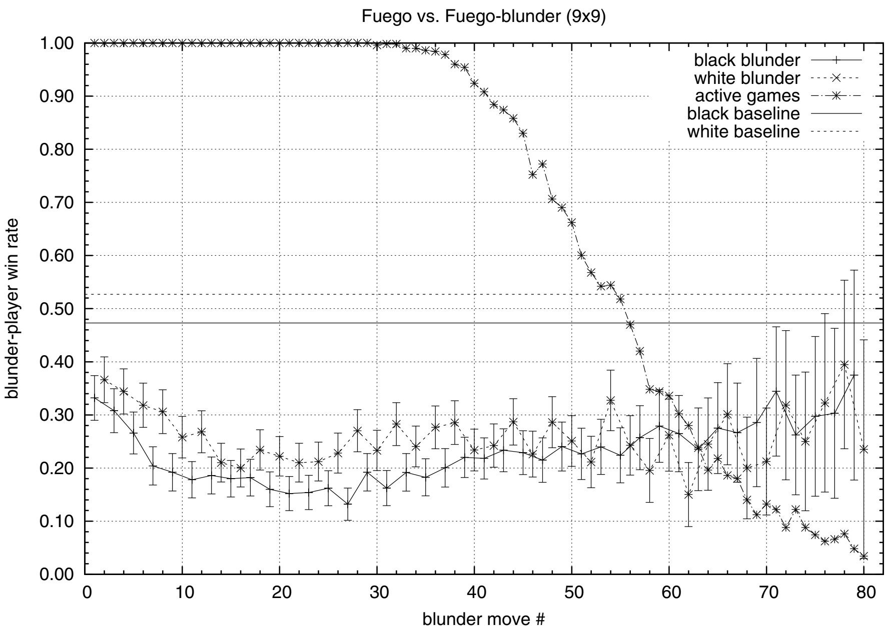


Fig. 1. FUEGO-blunder performance, 1000 sim./move, 9×9 board

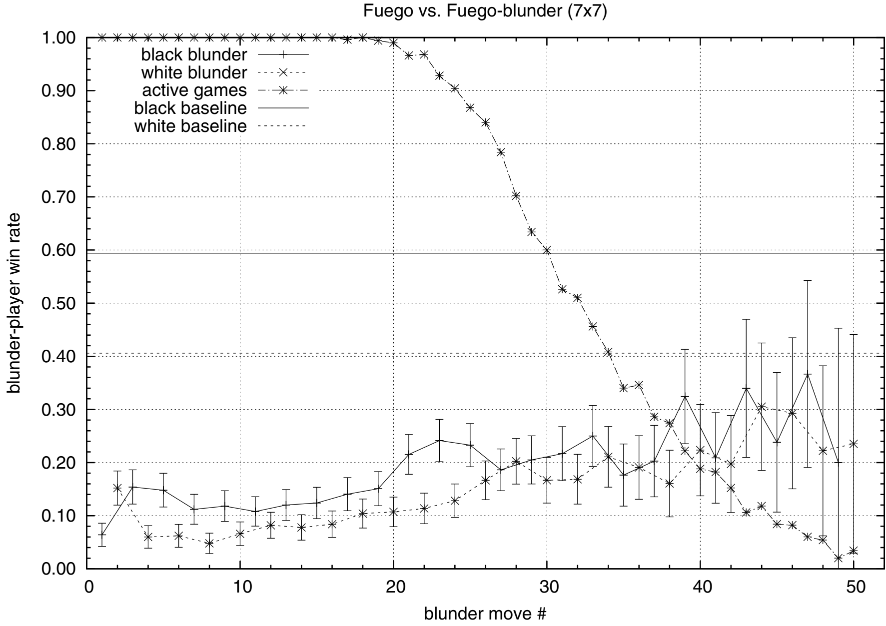


Fig. 2. FUEGO-blunder performance, 1000 sim./move, 7×7 board

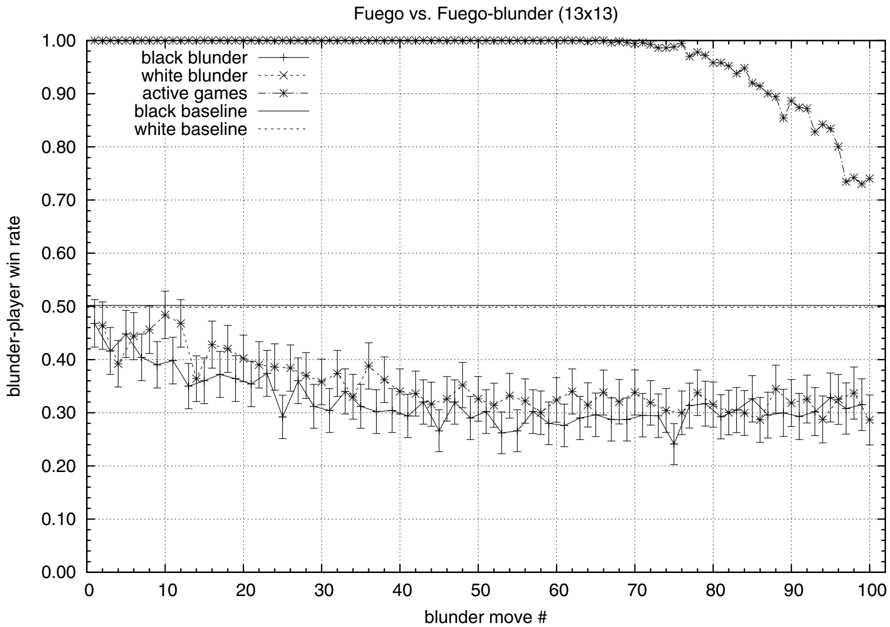


Fig. 3. FUEGO-blunder performance, 1000 sim./move, 13×13 board

board the komi of 7.5 is in White's favor in games between programs. The move 1 (black) blunder-player has a 33% win rate, so the blunder cost of this move is $47 - 33 = 14\%$. This win rate indicates that, for roughly 1/3 of the 49 possible 9×9 opening moves, FUEGO wins in self-play from that opening. At move 56 the active game rate is below 0.5, so for this move more than half of the 500 trial games finished before the blunder-player had a chance to make its move-56 blunder; as mentioned earlier, each data value is computed only on the games still active at that the time of the scheduled blunder, so the error bar here is larger than at moves made when all 500 trial games were active.

In the early game, the 9×9 blunder-player win rate is relatively high, suggesting that FUEGO's play here is relatively weak and/or that the number of available winning moves is relatively high. By move 11 the blunder cost is about $47 - 18 = 29\%$, about double the move-1 blunder cost.

3.2 Go on Other Board Sizes

Compare the 9×9 data with the 7×7 data in Figure 2 and the 13×13 data in Figure 3. For the 7×7 data, the number of simulations per move is the same but the number of available moves per position is smaller (than for 9×9). Thus we expect FUEGO to be stronger here, and this seems to be the case, as blunder costs are higher than for 9×9 Go. FUEGO may play 7×7 Go close to perfectly, in which case the move-k blunder-player win rate is close to the move-k available-winning-move rate.

By contrast, for the 13×13 data, with a larger number of moves per position, we expect FUEGO to be weaker. Again, this appears to be the case, as blunder costs are lower. The move-1 blunder cost is under 4%, and it is only by about move 50 that blunder cost is about 25%. For some reason, even though black and white baseline win rates are within 1% of each other, black blunders are typically more costly than white blunders. This is a topic for further study.

4 Blunder Analysis of MoHex

Hex is the classic connection game created by Piet Hein in 1942 [13] and John Nash around 1948 [18]. The players alternate turns, trying to connect their two sides. Figure 4 shows a game won by Black.

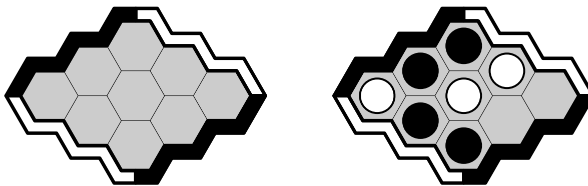


Fig. 4. An empty 3×3 Hex board and a game won by Black

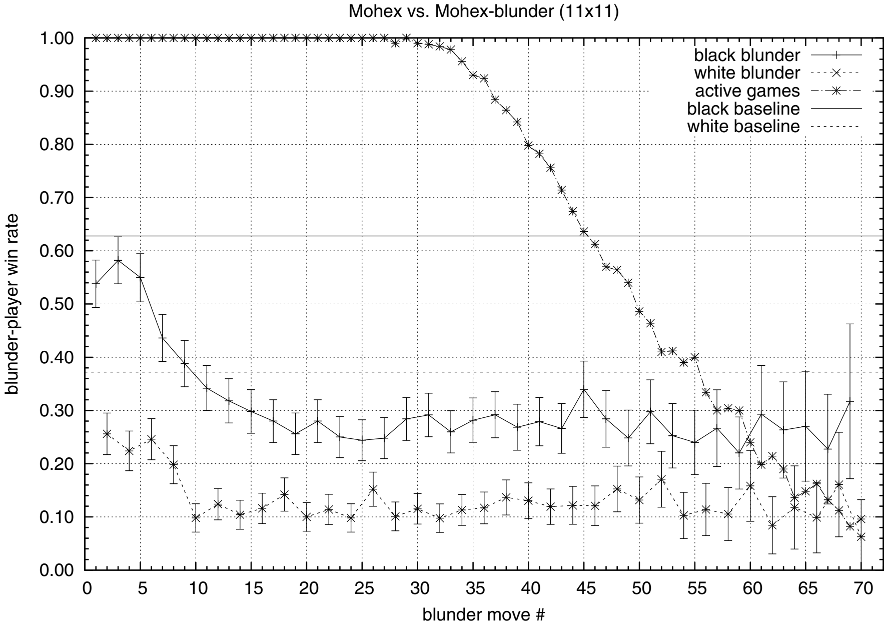


Fig. 5. MoHex-blunder performance, 1000 sim./move, 11×11 board

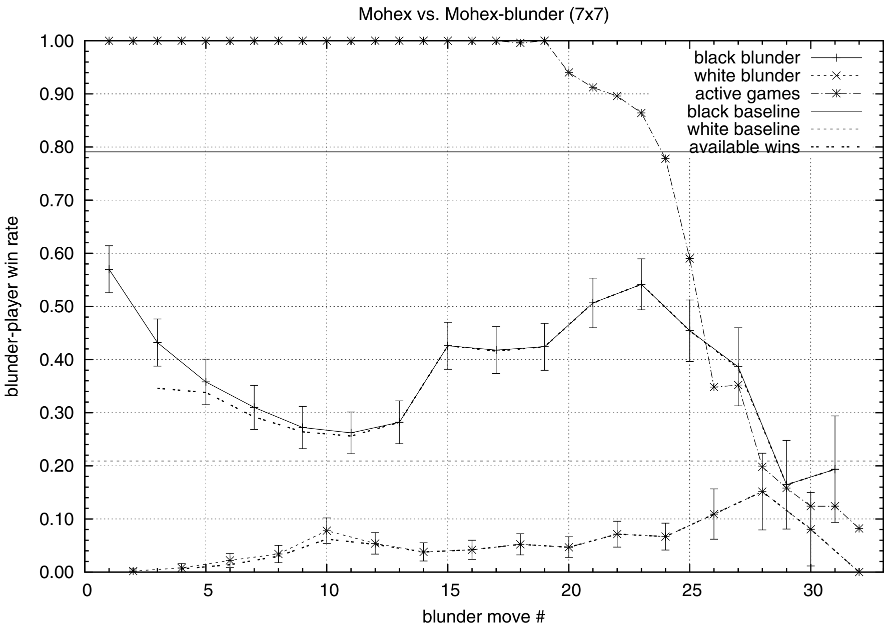


Fig. 6. Blunder-MoHex performance, 1000 sim./move, 7×7 board, also showing available-winning-move rate

Hex is simpler than Go in some aspects: stones never move once played, checking the winning condition is easy, and the game cannot end in a draw. On $n \times n$ boards the first player has a winning strategy, but no explicit such strategy is known for any $n > 9$, and solving arbitrary positions is Pspace-complete. (For more on Hex, including more on these results, see [6] or [11].) For this reason, Hex is often used as a test bed for algorithmic development.

MoHex is the open source MCTS Hex program originally developed by the University of Alberta Computer Hex group, led by Ryan Hayward and including Broderick Arneson and Philip Henderson [5,3]. MoHex is built on top of the Fuego Monte Carlo tree search framework. Its main game-specific ingredients are a virtual connection[1] engine and an inferior cell[2] engine [3]. MoHex won the gold medal for (11×11) Hex at the 2009 and 2010 ICGA Computer Games Olympiads [1,2].

The parameters for Hex experiments were in general the same as for the Go experiments. Unless otherwise noted, MoHex ran 1000 MCTS simulations per move.

4.1 11×11 Hex

Figure 5 shows the effect of MoHex blunders on the 11×11 board. The baselines show first player (black) and second player (white) win rates of 62.8% and 37.2% respectively.[3] For some reason the white move-$k+1$ blunder cost is for small k slightly more than the black move-k blunder cost, but becomes less as k increases. This is a topic for further study.

4.2 7×7 Hex and Available-Winning-Move Rate

Although $n \times n$ Hex is a first player win, the baseline black win rate here is only 79%, rather than the 100% achievable by a perfect player. So, with 1000 simulations per move, MoHex is far from perfect, even on this small board.

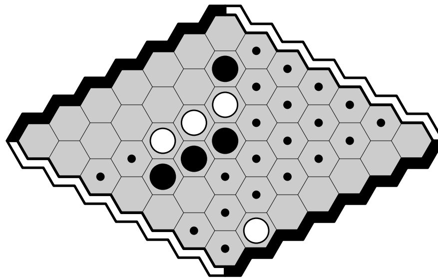


Fig. 7. A 7×7 Hex position with all winning black moves

[1] A virtual connection is a point-to-point 2nd-player connection strategy, i.e., the player can force the connection even if the opponent moves first.

[2] An inferior cell is one that can be pruned in the search for a winning move.

[3] This is perhaps partly due to MoHex arbitrarily assigning so-called dead cells (cells which are not in a minimal winning path-completion for either player) always to the black player in the inferior cell engine.

However, the move-1 blunder win rate is about 57±4%. This is within error of what is expected from a perfect player, since exactly 27/49≃0.55 of the possible 7×7 opening moves are winning [12], so any errors MoHex is making in these games are not having much effect.

State-of-the-art Hex solvers can easily solve arbitrary 7×7 positions [14,4]. So, in our 7×7 Hex experiment, for each position where the blunder player was about to move, we ran an exact solver to find the number of available winning moves at that point. For example, Figure 7 shows an 8-stone 7×7 position and all 21 winning moves. If Black blunders in this position, Black has a 21/41 chance of selecting a winning move.

In addition to the usual blunder data, Figure 6 shows the available-winning-move rates as two curves, one for each color. These curves are difficult to see, as from move 12 they coincide almost exactly with the respective blunder rates, suggesting that MoHex is playing most of these positions perfectly.

4.3 9×9 Hex and Playing Strength

Figures 8 and 9 show data on the same board size with two players of slightly different strength. We expect that in general the stronger player will have higher blunder cost, but in fact the opposite seems to occur here. This is a topic for further study.

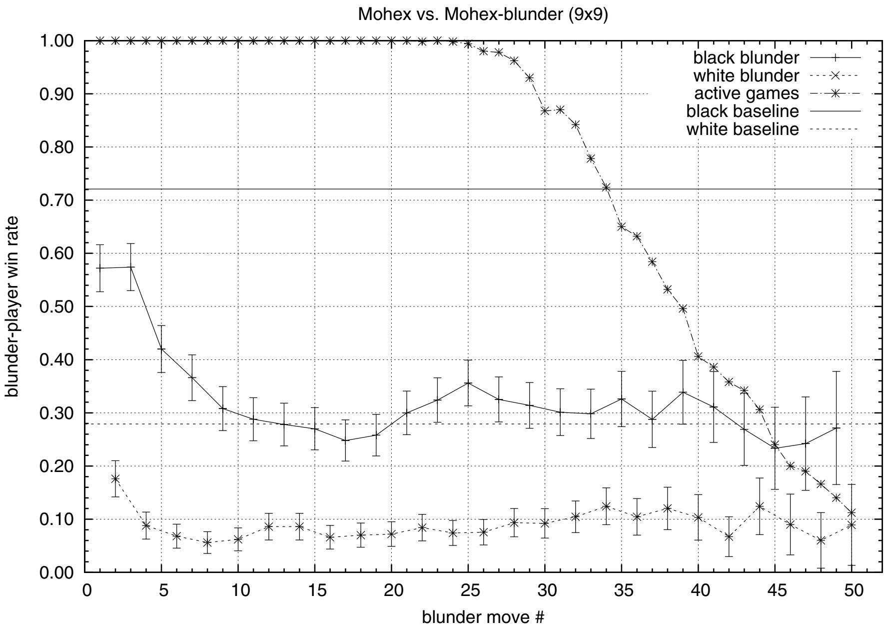


Fig. 8. MoHex-blunder performance, 1000 sim./move, 9×9 board

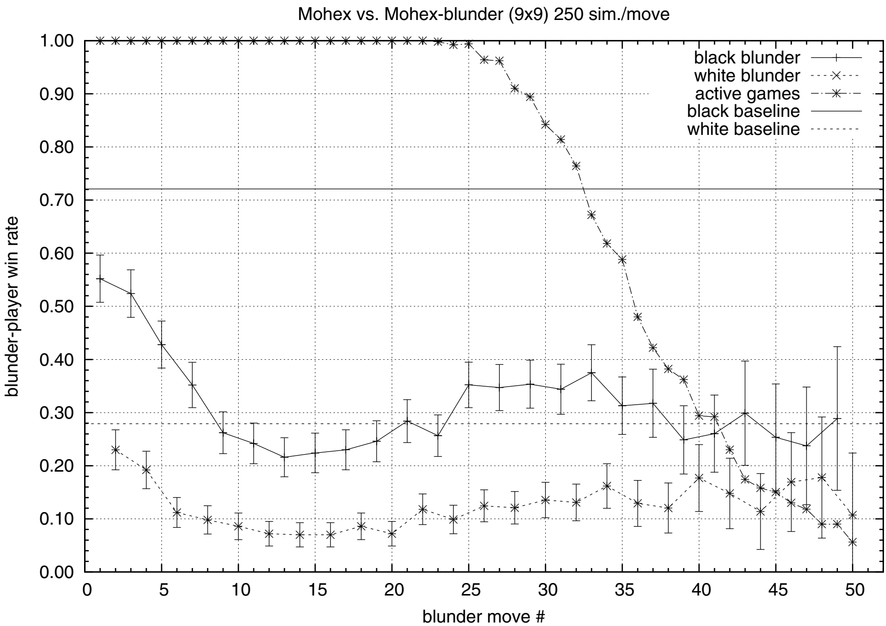


Fig. 9. MoHex-blunder performance, 250 sim./move, 9×9 board

5 Conclusions

We have shown that blunder analysis can offer insight into player performance and game difficulty. Our results with Fuego and MoHex suggest that when the players are strong and the board size is relatively small, the blunder rate corresponds closely with the fraction of available winning moves. We conjecture that this holds for other games and players. We propose blunder analysis as a reasonable measure of perfect play in general.

One possible use of blunder analysis is as a tool for aiding in game time management, i.e., deciding how much processing time to spend on move selection at various points in a game. We leave this as a topic for further study.

Acknowledgments. We thank the referees for their helpful comments. We thank NSERC Discovery (Hayward, Müller), NSERC USRA (Brausen), Prof. Jonathan Schaeffer and Alberta iCORE (Spies), UofA India Internship (Qadir), and the UofA GAMES group for their support of this project.

References

1. Arneson, B., Hayward, R.B., Henderson, P.: MoHex Wins Hex Tournament. ICGA 32(2), 114–116 (2009)
2. Arneson, B., Hayward, R.B., Henderson, P.: MoHex Wins Hex Tournament. ICGA 33(2), 181–186 (2010)

3. Arneson, B., Hayward, R.B., Henderson, P.: Monte Carlo Tree Search in Hex. IEEE Transactions on Computational Intelligence and AI in Games. Special issue on Monte Carlo Techniques and Computer Go 2(4), 251–257 (2010), www.cs.ualberta.ca/~hayward/publications.html
4. Arneson, B., Hayward, R.B., Henderson, P.: Solving Hex: Beyond Humans. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 1–10. Springer, Heidelberg (2011), www.cs.ualberta.ca/~hayward/publications.html
5. Arneson, B., Henderson, P., Hayward, R.B.: Benzene (2009-2011), http://benzene.sourceforge.net/
6. Browne, C.: Connection Games: Variations on a Theme. A.K. Peters, Wellesley, Massachusetts (2005)
7. University of Electro-Communications, J.: The Fourth Computer Go UEC Cup (2010), http://jsb.cs.uec.ac.jp/~igo/past/2010/eng/
8. Enzenberger, M., Müller, M., Arneson, B., Segal, R.: Fuego – an Open-Source Framework for Board Games and Go Engine based on Monte Carlo Tree Search. IEEE Transactions on Computational Intelligence and AI in Games. Special issue on Monte Carlo Techniques and Computer Go 2(4), 259–270 (2010)
9. Harper, D.: Online Etymology Dictionary (2001-2011), http://www.etymonline.com/
10. Haworth, G.M.: Reference fallible endgame play. ICGA 26(2), 81–91 (2003)
11. Hayward, R., van Rijswijck, J.: Hex and Combinatorics. Discrete Mathematics 306, 2515–2528 (2006)
12. 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. IFIP, vol. 263, pp. 261–278. Kluwer Academic Publishers, Boston (2003)
13. Hein, P.: Vil de laere Polygon? Politiken, December 26 (1942)
14. Henderson, P., Arneson, B., Hayward, R.B.: Solving 8×8 Hex. In: Proc. IJCAI, pp. 505–510 (2009)
15. International Computer Games Association: ICGA Tournaments (2011), http://www.grappa.univ-lille3.fr/icga/
16. Kroeker, K.L.: A New Benchmark for Artificial Intelligence. Communications of the ACM 54(8), 13–15 (2011)
17. Müller, M., Enzenberger, M., Arneson, B.: Fuego (2008-2011), http://fuego.sourceforge.net/
18. Nash, J.: Some games and machines for playing them. Tech. Rep. D-1164, RAND (February 1952)
19. Nau, D.S.: An investigation of the causes of pathology in games. Artificial Intelligence 19(3), 257–278 (1982)
20. Regan, K.W., Haworth, G.M.: Intrinsic chess rating. In: AAAI 2011: 25th AAAI Conf. on AI, pp. 834–839 (2011)

Understanding Distributions of Chess Performances

Kenneth W. Regan[1], Bartlomiej Macieja[2], and Guy M^{c}C. Haworth[3]

[1] Department of CSE, University at Buffalo, Amherst, NY 14260 USA
regan@buffalo.edu
[2] Warsaw, Poland
[3] School of Systems Engineering, University of Reading, UK
guy.haworth@bnc.oxon.org

Abstract. This paper studies the population of chess players and the distribution of their performances measured by Elo ratings and by computer analysis of moves. Evidence that ratings have remained stable since the inception of the Elo system in the 1970's is given in three forms: (1) by showing that the population of strong players fits a straightforward logistic-curve model without inflation, (2) by plotting players' average error against the FIDE category of tournaments over time, and (3) by skill parameters from a model that employs computer analysis keeping a nearly constant relation to Elo rating across that time. The distribution of the model's *Intrinsic Performance Ratings* can therefore be used to compare populations that have limited interaction, such as between players in a national chess federation and FIDE, and ascertain relative drift in their respective rating systems.

1 Introduction

Chess players form a dynamic population of varying skills, fortunes, and aging tendencies, and participate in zero-sum contests. A numerical rating system based only on the outcomes of the contests determines everyone's place in the pecking order. There is much vested interest in the accuracy and stability of the system, with significance extending to other games besides chess and potentially wider areas. Several fundamental questions about the system lack easy answers: How accurate are the ratings? How can we judge this? Have ratings inflated over time? How can different national rating systems be compared with the FIDE system? How much variation in performance is intrinsic to a given skill level?

This paper seeks statistical evidence beyond previous direct attempts to measure the system's features. We examine player-rating distributions across time since the inception of the Elo rating system by the World Chess Federation (FIDE) in 1971. We continue work by Haworth, DiFatta, and Regan [1,2,3,4] on measuring performance 'intrinsically' by the quality of moves chosen rather than the results of games. The models in this work have adjustable parameters that correspond to skill levels calibrated to the Elo scale. We have also measured aggregate error rates judged by computer analysis of entire tournaments, and plotted them against the Elo rating *category* of the tournament. Major findings of this paper extend the basic result of [4] that ratings have remained stable since the 1970's, contrary to the popular wisdom of extensive *rating inflation*. Section 5 extends that work to the Elo scale, while the other sections present independent supporting material. Related previous work [5,6,7] is discussed below.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 230–243, 2012.
© Springer-Verlag Berlin Heidelberg 2012

2 Ratings and Distributions

The Elo rating system, which originated for chess but is now used by many other games and sports, provides rules for updating ratings based on performance in games against other Elo-rated players, and for bringing new (initially 'unrated') players into the system. In chess, they have a numerical scale where 2800 is achieved by a handful of top players today, 2700 is needed for most highest-level tournament invitations, 2600 is a 'strong' grandmaster (GM), while 2500 is typical of most GM's, 2400 of International Masters, 2300 of FIDE Masters, and 2200 of masters in national federations. We emphasize that the ratings serve two primary purposes:

1. to indicate a position in the world ranking, and
2. to indicate a level of skill.

These two purposes lead to different interpretations of what it means for "inflation" to occur. According to view 1, 2700 historically meant what the neighborhood of 2800 means now: being among the very best, a true world championship challenger. As late as 1981, Anatoly Karpov topped the ratings at 2695, so no one had 2700, while today there are forty-five players 2700 and higher, some of whom have never been invited to an elite event. Under this view, inflation has occurred *ipso-facto*.

While view 2 is fundamental and has always had adherents, for a long time it had no reliable benchmarks. The rating system itself does not supply an intrinsic meaning for the numbers and does not care about their value: arbitrarily add 1000 to every figure in 1971 and subsequent initialization of new players, and relative order today would be identical. However, recent work [4] provides a benchmark to calibrate the Elo scale to games analyzed in the years 2006–2009, and finds that ratings fifteen and thirty years earlier largely correspond to the same benchmark positions. In particular, today's echelon of over forty 2700+ players all give the same or better statistics in this paper than Anatoli Karpov and Viktor Korchnoi in their prime. We consider that two further objections to view 2 might take the following forms.

(a) If Karpov and Korchnoi had access to today's computerized databases and more extensive opening publications, they would have played (say) 50 to 100 points higher—as Kasparov did as the 1980's progressed.
(b) Karpov and Korchnoi were supreme strategists whose strategic insight and *depth* of play does not show up in ply-limited computer analysis.

We answer (a) by saying we are concerned only with the quality of moves made on the board, irrespective of whether and how they are prepared. Regarding also (b) we find that today's elite make fewer clear mistakes than their forebears. This factor impacts skill apart from strategic depth. The model from [4] used in this paper finds a natural weighting for the relative importance of avoiding mistakes.

Our position in subscribing to view 2 is summed up as *today's players deserve their ratings*. The numerical rating should have a fixed meaning apart from giving a player's rank in the world pecking order. In subsequent sections we present the following evidence that there has been no inflation, and that the models used for our conclusions produce reasonable distributions of chess performances.

- The proportion of Master-level ratings accords exactly with what is predicted from the growth in population alone, without adjusting for inflation.
- A version, called AE for "average error," of the "average difference" (AD) statistic used by Guid and Bratko [5] (see also [6,7]) to compare world championship matches. An important scaling discovery leads to *Scaled Average Error* (SAE). Our work shows that tournaments of a given category have seen a fairly constant (S)AE over time.
- "Intrinsic Ratings" as judged from computer analysis have likewise remained relatively constant as a function of Elo rating over time—for this we refine the method by Regan and Haworth [4].
- Intrinsic Ratings for the world's top players have increased steadily since the mid-1800s, mirroring the way records have improved in many other sports and human endeavors.
- Intrinsic Performance Ratings (IPR's) for players in events fall into similar distributions as assumed for Tournament Performance Ratings (TPR's) in the rating model, with somewhat higher variance. They can also judge inflation or deflation between two rating systems, such as those between FIDE and a national federation much of whose population has little experience in FIDE-rated events.

The last item bolsters the Regan-Haworth model [4] as a reliable indicator of performance, and therefore enhances the significance of the third and fourth items.

The persistence of rating controversies after many years of the standard analysis of rating curves and populations calls to mind the proverbial elephant that six blind men are trying to picture. Our non-standard analyses may take the hind legs, but since they all agree, we feelm we understand the elephant. Besides providing new insight into distributional analysis of chess performances, the general nature of our tools allows application in other games and fields besides chess.

3 Population Statistics

Highlighted by the seminal work of de Solla Price on the metrics of science [8], researchers have gained an understanding of the growth of human expertise in various subjects. In an environment with no limits on resources for growth, de Solla Price showed that the rate of growth is proportional to the population,

$$\frac{dN}{dt} \sim aN, \tag{1}$$

which yields an exponential growth curve. For example, this holds for a population of academic scientists, each expected to graduate some number $a > 1$ of students as new academic scientists. However, this growth cannot last forever, as it would lead to a day when the projected number of scientists would be greater than the total world population. Indeed, Goodstein [9] showed that the growth of PhD's in physics produced each year in the United States stopped being exponential around 1970, and now remains at a constant level of about 1000.

The theory of the growth of a population under limiting factors has been successful in other subjects, especially in biology. Since the work by Verhulst [10] it has been

widely verified that in an environment with limited resources the growth of animals (for instance tigers on an island) can be well described by a logistic function

$$N(t) = \frac{N_{max}}{(1 + a(exp)^{-bt})} \qquad \text{arising from} \qquad \frac{dN}{dt} \sim aN - bN^2, \tag{2}$$

where bN^2 represents a part responsible for a decrease of a growth due to an overpopulation, which is quadratic insofar as every animal interacts, for instance fights for resources, with every other animal. We demonstrate that this classic model also describes the growth of the total number of chess players in time with a high degree of fit.

We use a minimum rating of 2203—which FIDE for the first three Elo decades rounded up to 2205—because the rating floor and the start rating of new players have been significantly reduced from 2200 which was used for many years.

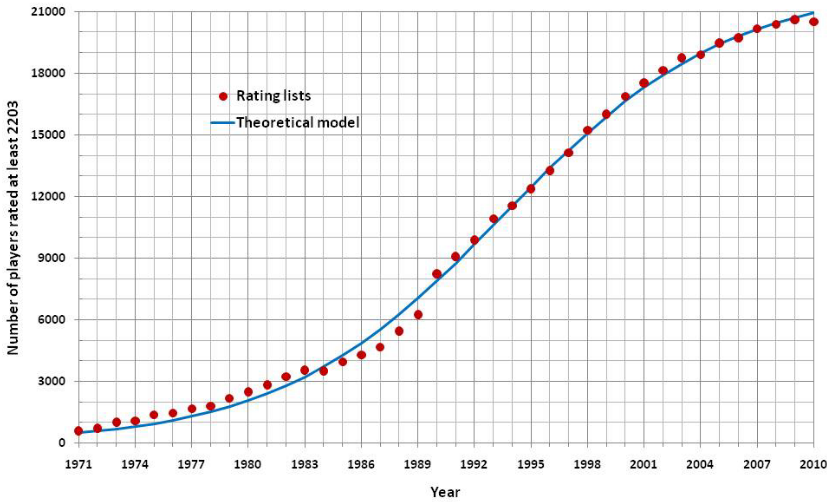


Fig. 1. Growth of number of players rated at least 2203 since 1971

Figure 1 shows the number of 2203+ rated players, and a curve obtained for some particular values of a, b, and N_{max}. Since there are many data points and only three parameters, the fit is striking. This implies that the growth of the number of chess players can be explained without a need to postulate inflation.

4 Average Error and Results by Tournament Categories

The first author has run automated analysis of almost every major event in chess history, using the program RYBKA 3 [11] to fixed reported depth 13 ply[1] in Single-PV mode.

[1] That RYBKA versions often report the depth as -2 or -1 in UCI feedback has fueled speculation that the true depth here is 16, while the first author finds it on a par in playing strength with some other prominent programs fixed to depths in the 17–20 range.

This mode is similar to how Guid and Bratko [5] operated the program CRAFTY to depth (only) 12, and how others have run other programs since. Game turns 1–8, turns where RYBKA reported a more than 3.00 advantage already at the previous move, and turns involved in repetitions are weeded out.

The analysis computations have included *all* round-robin events of Category 11 or higher, using all events given categories in the ChessBase Big 2010 database plus The Week In Chess supplements through TWIC 893 12/19/11. The categories are the average rating of players in the event taken in blocks of 25 points; for instance, category 11 means the average rating is between 2500 and 2525, while category 15 means 2600–2625.

For every move that is not equivalent to RYBKA's top move, the "error" is taken as the value of the present position minus the value after the move played. The errors over a game or player-performance or an entire tournament are summed and divided by the number of moves (those not weeded out) to make the "Average Error" (AE) statistic. Besides including moves 9–12 and using Rybka depth 13 with a $[-3.00, +3.00]$ evaluation range rather than CRAFTY depth 12 with a $[-2.00, +2.00]$ range, our statistic differs from [5] in not attempting to judge the "complexity" of a position, and in several incidental ways.

For large numbers of games, AD or AE seems to give a reasonable measure of playing quality, beyond relative ranking as shown in [6]. When aggregated for all tournaments in a span of years, the figures were in fact used to make scale corrections for the in-depth mode presented in the next section. When AE is plotted against the turn number, sharply greater error for turns approaching the standard Move 40 time control is evident; then comes a sharp drop back to previous levels after Move 41. When AE is plotted against the advantage or disadvantage for the player to move, in intervals of 0.10 or 0.05 pawns, a scaling pattern emerges. The AE for advantage 0.51–0.60 is almost double that for near-equality 0.01–0.10, while for -0.51 to -0.60 it is regularly more than double.

It would seem strange to conclude that strong masters play only half as well when ahead or behind by half a Pawn as even. Rather this seems to be evidence that human players perceive differences in value in proportion to the overall advantage for one side. This yields a log-log kind of scaling, with an additive constant that tests place close to 1, so we used 1. This is reflected in the definition of the scaled difference δ_i in Equation 3 below, since $1/(1 + |z|)$ in the body of a definite integral produces $\ln(1 + |z|)$. This produces *Scaled Average Error* (SAE).

Figure 2 shows AE (called R3 for "raw" and the 3.00 evaluation cutoff) and SAE (SC3), while Figure 3 shows how both figures increase markedly toward the standard Move 40 time control and then level off. For these plots the tournaments were divided into historical "eras" E5 for 1970–1984, E6 for 1985–1999, E7 for 2000–2009, and E8 for 2010–. The tournaments totaled 57,610 games, from which 3,607,107 moves were analyzed (not counting moves 1–8 of each game which were skipped) and over 3.3 million retained within the cutoff. Category 10 and lower tournaments that were also analyzed bring the numbers over 60,000 games and 4.0 million moves with over 3.7 million retained. Almost all work was done on two quad-core Windows PC's with analysis scripted via the Arena GUI v1.99 and v2.01.

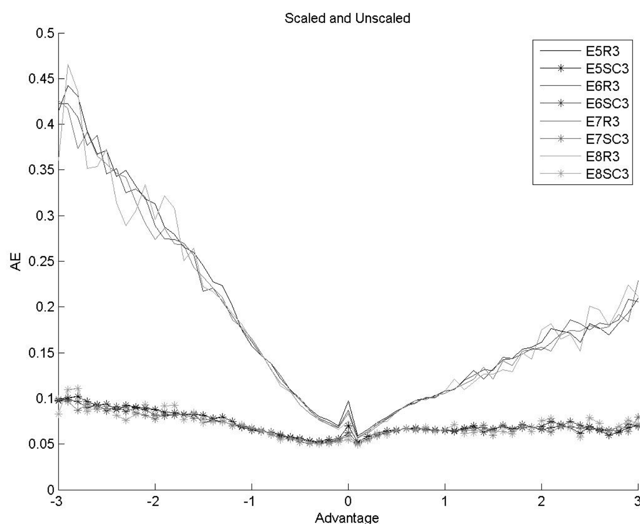


Fig. 2. Plot of raw AE vs. advantage for player to move, and flattening to SAE

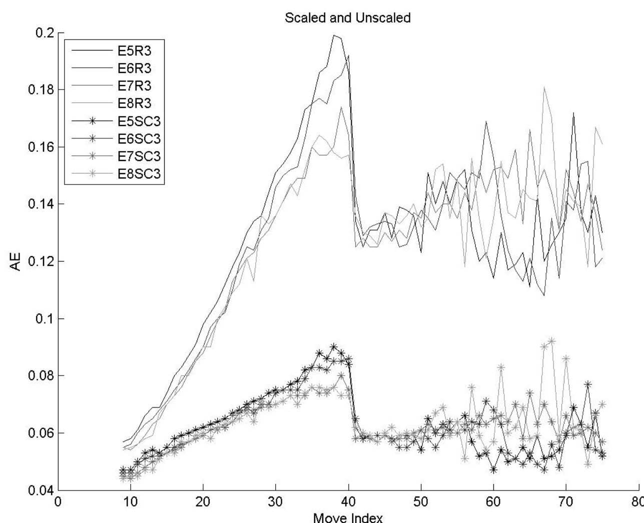


Fig. 3. Plot of AE and SAE by turn number

Figures 4 and 5 below graph SAE for all tournaments by year as a *four-year moving average*, the latter covering moves 17–32 only. The five lines represent categories 11–12 (FIDE Elo 2500–2549 average rating), 13–14 (2550–2599), 15–16 (2600–2649), 17–18 (2650–2699), and 19–20 (2700–2749). There were several category 21 events in 1996–2001, none in 2002–2006, and several 21 and 22 events since 2007; the overall averages of the two groups are plotted as X for 2001 and 2011. The lowest category has the highest SAE and therefore appears at the top.

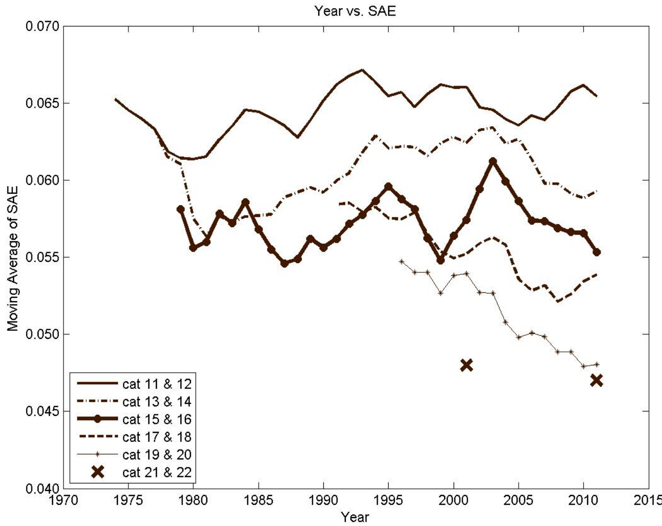


Fig. 4. SAE by tournament category, 4-yr. moving avg., 1971–2011

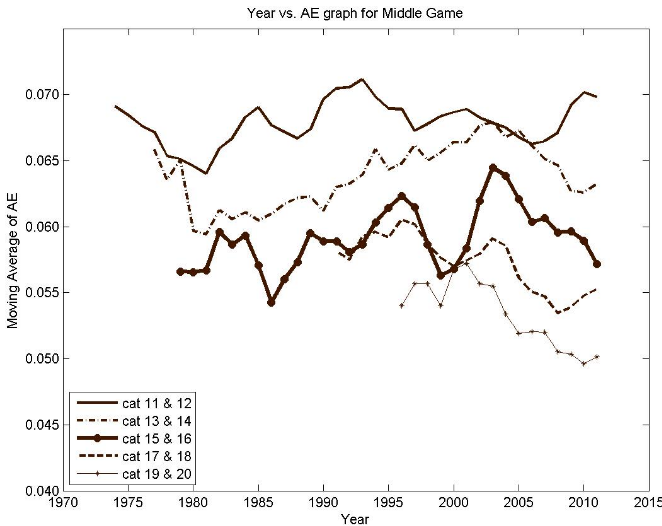


Fig. 5. SAE by category for moves 17–32 only, 4-yr. moving avg., 1971–2011

Despite yearly variations the graphs allow drawing two clear conclusions: (1) the categories do correspond to different levels of SAE, and (2) the lines by-and-large do not slope up to the right as would indicate inflation. Indeed, the downslope of SAE for categories above 2650 suggests some deflation since 1990. Since the SAE statistic depends on how tactically challenging a game is, and thus does not indicate skill by itself, we need a more intensive mode of analysis in order to judge skill directly.

5 Intrinsic Ratings over Time

Haworth [1,2] and with DiFatta and Regan [3,12,4] developed models of fallible decision agents that can be trained on players' games and calibrated to a wide range of skill levels. Their main difference from [5,6,7] is the use of Multi-PV analysis to obtain authoritative values for all reasonable options, not just the top move(s) and the move played. Thus each move is evaluated in the full context of available options. The paper [6] gives evidence that for relative rankings of players, good results can be obtained even with relatively low search depths, and this is confirmed by [7]. However, we argue that for an intrinsic standard of quality by which to judge possible rating drift, one needs greater depth, the full move context, and a variety of scientific approaches. The papers [3,12] apply Bayesian analysis to characterize the performance of human players using a spectrum of *reference fallible agents*. The work reported in [4] and this paper uses a method patterned on multinomial Bernoulli trials, and obtains a corresponding spectrum.

The *scaling* of AE was found important for quality of fit, and henceforth AE means SAE. It is important to note that SAE from the last section does not directly carry over to intrinsic ratings in this section, because here we employ the full move analysis of Multi-PV data. They may be expected to correspond in large samples such as all tournaments in a range of years for a given category, but here we are considering smaller samples from a single event or a single player in a single event, and at this stage we are studying those with more intensive data. What we do instead is use statistical fits of parameters called s, c to generate *projections* AE_e for every position, and use the aggregate projected AE_e on a reference suite of positions as our "standard candle" to index to the Elo scale.

We also generate projected standard deviations, and hence projected confidence intervals, for AE_e (and also the first-move match statistic MM_e) as shown below. This in turn yields projected confidence intervals for the intrinsic ratings. Preliminary testing with randomly-generated subsets of the training data suggest that the actual deviations in real-world data are bounded by a factor of 1.15 for the MM statistic and 1.4 for AE, and these are signified by a subscripted a for 'actual' in tables below. The projections represent the ideal case of zero modeling error, so we regard the difference shown by the tests as empirical indication of the present level of modeling error.

Models of this kind function in one direction by taking in game analyses and using statistical fitting to generate values of the skill parameters to indicate the intrinsic level of the games. They function in the other direction by taking pre-set values of the skill parameters and generating a probability distribution of next moves by an agent of that skill profile. The defining equation of the particular model used in [4], relating the probability p_i of the i-th alternative move to p_0 for the best move and its difference in value, is

$$\frac{\log(1/p_i)}{\log(1/p_0)} = e^{-\left(\frac{\delta}{s}\right)^c}, \qquad \text{where} \qquad \delta_i = \int_{v_i}^{v_0} \frac{1}{1+|z|} dz. \tag{3}$$

Here when the value v_0 of the best move and v_i of the i-th move have the same sign, the integral giving the scaled difference simplifies to $|\log(1+v_0) - \log(1+v_i)|$. Note that this employs the empirically-determined scaling law from the last section.

The skill parameters are called s for "sensitivity" and c for "consistency" because s when small can enlarge small differences in value, while c when large sharply cuts down the probability of poor moves. The equation solved directly for p_i becomes

$$p_i = p_0^\alpha \qquad \text{where} \qquad \alpha = e^{-\left(\frac{\delta}{s}\right)^c}. \tag{4}$$

The constraint $\sum_i p_i = 1$ thus determines all values. By fitting these derived probabilities to actual frequencies of move choice in training data, we can find values of s and c corresponding to the training set.

Each Elo century mark 2700, 2600, 2500, ... is represented by the training set comprising all available games under standard time controls in round-robin or small-Swiss (such as no more than 54 players for 9 rounds) in which both players were rated within 10 points of the mark, in the three different time periods 2006–2009, 1991–1994, and 1976–1979. In [4], it was observed that the computed values of c stayed within a relatively narrow range, and gave a good linear fit to Elo rating by themselves. Thus it was reasonable to impose that fit and then do a single-parameter regression on s. The "central s, c artery" created this way thus gives a clear linear relation to Elo rating.

Here we take a more direct route by computing from any (s, c) a single value that corresponds to an Elo rating. The value is the *expected error per move* on the union of the training sets. We denote it by AE_e, and note that it, the expected number MM_e of matches to the computer's first-listed move, and projected standard deviations for these two quantities, are given by these formulas:

$$\begin{aligned} MM_e &= \sum_{t=1}^T p_{0.t}, & \sigma_{MM_e} &= \sqrt{\sum_{t=1}^T p_{0,t}(1-p_{0,t})} \\ AE_e &= \frac{1}{T}\sum_{t=1}^T \sum_{i\geq 1} p_{i,t}\delta_{i,t}, & \sigma_{AE_e} &= \sqrt{\frac{1}{T}\sum_{t=1}^T \sum_{i\geq 1} p_{i,t}(1-p_{i,t})\delta_{i,t}}. \end{aligned} \tag{5}$$

The first table gives the values of AE_e that were obtained by first fitting the training data for 2006–09, to obtain s, c, then computing the expectation for the union of the training sets. It was found that a smaller set R of moves comprising the games of the 2005 and 2007 world championship tournaments and the 2006 world championship match gave identical results to the fourth decimal place, so R was used as the fixed *reference set*.

Table 1. Correspondence between Elo rating from 2006–2009 and projected Average Error

Elo	2700	2600	2500	2400	2300	2200
AE_e	.0572	.0624	.0689	.0749	.0843	.0883

A straightforward linear fit then yields the rule to produce the Elo rating for any (s, c), which we call an "Intrinsic Performance Rating" (IPR) when the (s, c) are obtained by analyzing the games of a particular event and player(s).

$$\text{IPR} = 3571 - 15413 \cdot AE_e. \tag{6}$$

This expresses, incidentally, that at least from the vantage of the RYBKA 3 run to reported depth 13, perfect play has a rating under 3600. This is reasonable when one

considers that if a 2800 player such as Vladimir Kramnik is able to draw one game in fifty, the opponent can never have a higher rating than that.

Using equation (6), we reprise the main table from [4], this time with the corresponding Elo ratings from the above formulas. The left-hand side gives the original fits, while the right-hand side corresponds to the "central artery" discussed above. The middle of the table is our first instance of the following procedure for estimating confidence intervals for the IRP derived from any test set.

1. Do a regression on the test set T to fit s_T, c_T.
2. Use s_T, c_T to project AE_e on the reference set R (not on T), and derive IPR$_T$ via equation (6).
3. Use s_T, c_T on the test set T only to project $\sigma_T = \sigma_{AE_e}$.
4. Output $[\text{IPR}_T - 2\sigma_T, \text{IPR}_T + 2\sigma_T]$ as the proposed "95%" confidence interval.

As noted toward the start of this section, early testing suggests replacing σ_T by $\sigma_a = 1.4\sigma_T$ to get an "actual" 95% confidence interval given the model as it stands. Hence, we show both ranges.

In this case, the test sets T are the training sets themselves for the Elo century points in three different four-year intervals. These give the results in Table 2.

Table 2. Elo correspondence in three four-year intervals

Elo	s	c	IPR	$2\sigma_e$ range	$2\sigma_a$ range	#moves	c_{fit}	s_{fit}	IPR$_{fit}$
2006–2009									
2700	.078	.502	2690	2648–2731	2632–2748	7,032	.513	.080	2698
2600	.092	.523	2611	2570–2652	2553–2668	7,807	.506	.089	2589
2500	.092	.491	2510	2480–2541	2468–2553	16,773	.499	.093	2528
2400	.098	.483	2422	2393–2452	2381–2464	20,277	.492	.100	2435
2300	.108	.475	2293	2257–2328	2243–2342	17,632	.485	.111	2304
2200	.123	.490	2213	2170–2257	2153–2274	11,386	.478	.120	2192
2100	.134	.486	2099	2048–2150	2028–2170	9,728	.471	.130	2072
2000	.139	.454	1909	1853–1966	1830–1989	9,471	464	.143	1922
1900	.159	.474	1834	1790–1878	1769–1893	16,195	.457	.153	1802
1800	.146	.442	1785	1741–1830	1723–1848	15,930	.450	.149	1801
1700	.153	.439	1707	1642–1772	1616–1798	8,429	.443	.155	1712
1600	.165	.431	1561	1496–1625	1470–1651	9,050	.436	.168	1565
1991–1994									
2700	.079	.487	2630	2576–2683	2555–2704	4,954	.513	.084	2659
2600	.092	.533	2639	2608–2670	2596–2682	13,425	.506	.087	2609
2500	.098	.500	2482	2453–2512	2441–2524	18,124	.499	.092	2537
2400	.101	.484	2396	2365–2426	2353–2438	19,968	.492	.103	2406
2300	.116	.480	2237	2204–2270	2191–2284	20,717	.485	.117	2248
2200	.122	.477	2169	2136–2202	2123–2215	21,637	.478	.122	2173
1976–1979									
2600	.094	.543	2647	2615–2678	2602–2691	11,457	.506	.087	2609
2500	.094	.512	2559	2524–2594	2509–2609	11,220	.499	.091	2547
2400	.099	.479	2397	2363–2431	2350–2444	16,635	.492	.103	2406
2300	.121	.502	2277	2240–2313	2226–2328	15,284	.485	.116	2257

Table 3. Intrinsic Ratings of Category 21 and higher standard tournaments

Event	cat: Elo	IPR	$2\sigma_e$ range	$2\sigma_a$ range	IPR-Elo	#moves
Las Palmas 1996	21: 2756	2697	2612–2781	2579–2815	-59	1,760
Linares 1998	21: 2752	2715	2651–2780	2625–2805	-37	2,717
Linares 2000	21: 2751	2728	2645–2810	2612–2843	-23	1,636
Dortmund 2001	21: 2755	2752	2760–2834	2637–2866	-3	1,593
Mexico 2007	21: 2751	2708	2647–2769	2623–2793	-43	3,213
Morelia-Linares 2008	21: 2755	2855	2808–2903	2789–2922	+100	3,453
Nanjing 2008	21: 2751	2766	2691–2842	2660–2873	+15	1,936
Bilbao GSF 2008	21: 2768	2801	2731–2872	2702–2900	+33	2,013
Linares 2009	21: 2755	2750	2696–2803	2675–2825	-5	3,830
Sofia M-Tel 2009	21: 2754	2711	2626–2795	2592–2829	-51	1,937
Nanjing 2009	21: 2763	2715	2644–2785	2616–2814	-48	2,192
Moscow Tal Mem. 2009	21: 2763	2731	2663–2800	2635–2827	-32	2,706
Linares 2010	21: 2757	2681	2607–2756	2577–2786	-76	2,135
Nanjing 2010	21: 2766	2748	2674–2821	2645–2850	-18	1,988
Shanghai 2010	21: 2759	2829	2727–2931	2686–2972	+70	920
Bilbao 2010	22: 2789	2904	2822–2987	2788–3020	+115	1,060
Moscow Tal Mem. 2010	21: 2757	2690	2629–2750	2604–2775	-67	3,493
Bazna 2011	21: 2757	2750	2675–2825	2645–2855	-7	1,885
Sao Paulo/Bilbao 2011	22: 2780	2626	2539–2713	2504–2748	-154	1,998
Moscow Tal Mem. 2011	22: 2776	2807	2755–2860	2734–2881	+31	3,401
Averages	21: 2761	2748			-13	2,293
Weighted by moves	21: 2760	2745			-15.6	
Aggregate run, all moves	21: 2760	2744	2729–2760	2722–2766	-16	45,870

Table 4. Some other events, for comparison to Table 3

Event	cat: Elo	IPR	$2\sigma_e$ range	$2\sigma_a$ range	IPR-Elo	#moves
Montreal 1979	15: 2622	2588	2534–2642	2513–2663	-34	4,732
Linares 1993	18: 2676	2522	2469–2574	2449–2595	-154	6,129
Linares 1994	18: 2685	2517	2461–2574	2438–2596	-168	5,536
Dortmund 1995	17: 2657	2680	2615–2744	2589–2770	+23	2,459
Dortmund 1996	18: 2676	2593	2518–2667	2489–2697	-83	2,796
Dortmund 1997	18: 2699	2639	2569–2709	2541–2737	-60	2,583
Dortmund 1998	18: 2699	2655	2579–2732	2548–2762	-44	2,284
Dortmund 1999	19: 2705	2749	2655–2844	2617–2882	+44	1,364
Sarajevo 1999	19: 2703	2664	2592–2737	2563–2766	+19	2,755
San Luis 2005	20: 2738	2657	2597–2716	2574–2740	-81	3,694
Corus 2006	19: 2715	2736	2693–2779	2676–2797	+21	5,800
Sofia M-Tel 2006	20: 2744	2744	2678–2810	2651–2836	0	2,197
Corus 2007	19: 2717	2763	2716–2811	2697–2829	+46	5,095
Sofia M-Tel 2007	19: 2725	2576	2482–2670	2445–2708	-149	2,184
Sofia M-Tel 2008	20: 2737	2690	2605–2775	2571–2809	-47	1,869
London Classic 2010	20: 2725	2668	2594–2742	2565–2771	-57	2,312

The entries vary around the Elo century marks, as is to be expected from a linear fit. Some points in the 1600–2100 range are anomalous, and this may owe to various factors pertaining to the quality of the games. Only the Elo 2200 through 2700 data for 2006–2009 were used in the linear fit for the ratings. Of course, there is error from the regression, but we do not know whether it adds to or mitigates the estimates σ_{AE_e} of placement of the linear regression points. For uniformity with later performance testing, we show only the latter error here. Despite these elements of uncertainty, the table still

Table 5. Comparison of FIDE and CFC ratings, TPR's, and IPR's for 2011 Canadian Open

Name	Can R	FIDE R	TPR	IPR	IPR-TPR	$2\sigma_e$ range	$2\sigma_a$ range	#moves
Arencibia	2537	2476	2745	2723	-22	2491–2956	2398–3049	273
Benjamin	2641	2553	2688	2412	-276	2196–2629	2110–2715	373
Bluvshtein	2634	2611	2622	2533	-89	2323–2744	2239–2828	316
Bojkov	2544	2544	2595	2154	-441	1765–2543	1610–2698	219
Calugar	2437	2247	2144	2301	+157	2091–2512	2007–2596	327
Cheng	2500	2385	2661	2728	+67	2502–2954	2411–3044	297
Cummings	2459	2350	2473	2833	+360	2683–2983	2623–3043	322
Fedorowicz	2508	2454	2422	2390	-32	2088–2692	1967–2813	199
Gerzhoy	2647	2483	2622	2963	+341	2802–3124	2738–3189	211
Golod	2576	2582	2582	2638	+56	2376–2899	2272–3003	218
Hebert	2486	2414	2519	2789	+270	2598–2979	2522–3055	285
Krnan	2470	2390	2651	2694	+43	2488–2900	2405–2982	266
Krush	2578	2487	2539	2497	-42	2217–2717	2189–2805	316
Meszaros	2409	2418	2278	2413	+133	2219–2607	2141–2684	337
Mikhalevski	2664	2569	2519	2616	+96	2412–2820	2330–2902	248
Milicevic	2400	2288	2352	2113	-240	1799–2426	1674–2552	214
Mulyar	2422	2410	2412	2636	+224	2483–2788	2422–2849	378
Noritsyn	2597	2425	2563	2394	-171	2166–2621	2075–2713	286
Pechenkin	2408	2297	2309	2648	+339	2439–2857	2355–2940	311
Perelshteyn	2532	2534	2650	2629	-21	2425–2833	2343–2915	258
Perez Rod'z	2467	2467	2676	2627	-49	2321–2933	2198–3056	195
Plotkin	2411	2243	2260	2715	+455	2570–2861	2512–2919	330
Regan	2422	2409	2268	2525	+257	2323–2728	2242–2809	356
Rozentalis	2614	2571	2666	2721	+55	2528–2913	2452–2990	291
Sambuev	2739	2528	2571	2677	+106	2499–2855	2428–2926	400
Samsonkin	2532	2378	2707	2535	-172	2267–2802	2159–2910	233
Sapozhnikov	2424	2295	2480	2404	-76	2203–2605	2122–2685	341
Shabalov	2618	2577	2549	2639	+90	2417–2861	2328–2590	262
Thavandiran	2447	2320	2607	2622	+15	2360–2884	2255–2989	254
Yoos	2439	2373	2289	1939	-350	1607–2271	1474–2404	268
Zenyuk	2429	2222	2342	2790	+448	2606–2975	2532–3049	229
Averages	2516	2429	2508	2558	+50			
Std. Dev.	92		157	218				
Whole event:	149					Restricted to	FIDE-rated	players: 115
Average	2144		2142	2117		2203	2211	2139
Std. Dev.	258		261	379		345	229	220
Wtd. avgs.						IPR	CanR	FIDE R
By games	2156		2154	2134		2219	2221	2147
By moves	2173		2172	2161		2242	2236	2161

supports a conclusion of no overall inflation. Because the fit was done with data from 2006–2009 only, inflation would show up as, for instance, 2600- and 2500-rated players from earlier years having higher IPR's than players with those ratings today.

Further support for our positions comes from IPR's of entire tournaments. Table 3 shows all twenty Category 21 or higher round-robin tournaments ever played under standard time controls, while Table 4 shows some others for comparison.

The IPR's are on-balance below the tournament average ratings, but the latter's aggregate is just within the narrower confidence interval of the aggregate IPR. The regressions are *not* linear, so the parity of the aggregate run with the weighted average is notable. The comparison events are selective but still show no inflationary trend.

6 Distributions of Performances

Our final experiment analyzed all 624 available games from 647 played at the 2011 Canadian Open, including all by players with FIDE ratings 2400 and above, which form an unbiased sample. Table 5 shows the IPR's and compares them to Chess Federation of Canada ratings before and after the event, FIDE ratings before, and the tournament performance ratings (TPR's) based on the CFC ratings. The final two columns are the confidence intervals for the IPR alone. The final rows summarize the sample, the whole event (152 players minus 3 early withdrawals leaving 149), and the whole event weighted by number of games played and number of analyzed moves. The bottom-right restricts to the 115 players who had FIDE ratings before the event. From the results we may conclude the following.

1. The IPR's have similar overall average to the Canadian ratings, especially under weighting by games or moves.
2. FIDE ratings of Canadian players are deflated relative to apparent skill. This is commonly believed to be due to a lack of playing opportunities in FIDE-rated events.
3. The IPR's have higher deviations from their own mean than the TPR's.
4. The IPR's have large deviation, and yet several TPR's fall outside even the 2.8-sigma range. This may constrain the usefulness of the IPR as an estimator of the TPR.

7 Conclusions

We have shown multiple, separate, and novel pieces of evidence that the Elo system employed by FIDE has remained stable in relation to intrinsic skill level. We have shown that the population of master-level players closely fits a model that has an important scientific pedigree, under conditions of no inflation. We have shown that ratings as reflected in tournament categories have no overall inflationary trend relative to two measures of skill, the simple AE statistic on a large scale embracing (nearly) all tournaments with at least 2500 average rating since 1971, and the more-intensive IPR statistic for some tournaments. We have also furthered the correspondence between Elo century marks and our model's fitted skill parameters shown in [4]. The IPR statistic is the weightiest evidence, but it is important that the other factors give it independent support. Given this stability in the FIDE system, we can promote the use of our tools in adjusting members of national federations with their own rating pools to the international scale.

We anticipate further development of the methods in this paper. It is possible that some rating systems being tested as alternatives to Elo in the recent *Kaggle* competitions sponsored by Sonas [13,14] may yield better correspondences to our models.

Acknowledgments. Foremost we thank the programmers of the Arena chess GUI for full scripting and recording of computer analysis, and those of TOGA II and RYBKA 3 for their engines and advice. Tamal Biswas collected data and prepared graphics. Support was provided by the UB Department of CSE and the University of Montreal for Jan.–June, 2009. Finally, we thank David Cohen and Hugh Brodie for providing gamescores of the entire 2011 Canadian Open, and the referees for helpful comments.

References

1. Haworth, G.: Reference fallible endgame play. ICGA Journal 26, 81–91 (2003)
2. Haworth, G.: Gentlemen, Stop Your Engines! ICGA Journal 30, 150–156 (2007)
3. DiFatta, G., Haworth, G., Regan, K.: Skill rating by Bayesian inference. In: Proceedings, 2009 IEEE Symposium on Computational Intelligence and Data Mining (CIDM 2009), Nashville, TN, March 30-April 2, pp. 89–94 (2009)
4. Regan, K., Haworth, G.: Intrinsic chess ratings. In: Proceedings of AAAI 2011, San Francisco (2011)
5. Guid, M., Bratko, I.: Computer analysis of world chess champions. ICGA Journal 29(2), 65–73 (2006)
6. Guid, M., Pérez, A., Bratko, I.: How trustworthy is Crafty's analysis of world chess champions? ICGA Journal 31(3), 131–144 (2008)
7. Guid, M., Bratko, I.: Using heuristic-search based engines for estimating human skill at chess. ICGA Journal 34(2), 71–81 (2011)
8. de Solla Price, D.J.: Science Since Babylon. Yale University Press (1961)
9. Goodstein, D.: The big crunch. In: Proceedings, 48th NCAR Symposium, Portland (1994)
10. Verhulst, P.F.: Notice sur la loi que la population poursuit dans son accroissement (1838)
11. Rajlich, V., Kaufman, L.: Rybka 3 chess engine (2007), http://www.rybkachess.com
12. Haworth, G., Regan, K., Di Fatta, G.: Performance and Prediction: Bayesian Modelling of Fallible Choice in Chess. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 99–110. Springer, Heidelberg (2010)
13. Sonas, J.: Chessmetrics (2011), http://www.chessmetrics.com
14. Sonas, J., Kaggle.com: Chess ratings: Elo versus the Rest of the World (2011), http://www.kaggle.com/c/chess

Position Criticality in Chess Endgames

Guy M^cC. Haworth[1] and Á. Rusz[2]

[1] School of Systems Engineering, University of Reading,
guy.haworth@bnc.oxon.org
[2] Budapest, Hungary

Abstract. Some 50,000 Win Studies in Chess challenge White to find an effectively unique route to a win. Judging the impact of less than absolute uniqueness requires both technical analysis and artistic judgment. Here, for the first time, an algorithm is defined to help analyse uniqueness in endgame positions objectively. The key idea is to examine how critical certain positions are to White in achieving the win. The algorithm uses sub-*n*-man endgame tables (EGTs) for both Chess and relevant, adjacent variants of Chess. It challenges authors of EGT generators to generalise them to create EGTs for these chess variants. It has already proved efficient and effective in an implementation for Starchess, itself a variant of chess. The approach also addresses a number of similar questions arising in endgame theory, games, and compositions.

1 Introduction

A Win Study in Chess is a composition in which White is challenged to win against Black's best defence. White's choice of move at each stage should be *effectively unique* even if not absolutely unique as in Sudoku or a crossword. Where there is more than one goal-compatible move, questions arise about the technical integrity and artistic quality of the study. The incidence of sub-7-man (s7m) mainline DTM-equi-optimal and DTM-sub-optimal moves in the HHDBIV corpus of over 76,000 studies [13] has been profiled [10] using Nalimov EGTs [3,18]. The comments of leading solvers, editors, and judges of studies make it clear that the *effective uniqueness question* is arguably the Grand Challenge for the study community. Beasley: "the detection of blind alleys in general is notoriously difficult." Roycroft: "When the depth difference is greater than two or three, one tends to shrug and move on to something else." Nunn: "detecting cycling moves can be easy in the case of a simple repetition or can be essentially impossible to do by hand in very complex cases."

Chess is a second-generation variant of a germinal board game and has inspired its own large family of variants [21,22]. However, it is still necessary to note that the Nalimov EGTs are in fact made for a variant of chess without castling.[1] The primary goal of chess variants is to provide an entertaining new challenge but the less radical variants also inform about chess itself. The creation of EGTs with restrictions on underpromotions [4,14,15,16] has, when compared with the standard EGTs, revealed

[1] Creating supplementary EGTs for positions with castling rights is in fact a small task.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 244–257, 2012.
© Springer-Verlag Berlin Heidelberg 2012

spectacular and essential underpromotions: see the positions *UPi* in Table 1, Figure 1, and the Appendix. The impact of the 50-move draw-claim rule has been noted [5] after computing EGTs for the chess variant $Chess_{50}$ in which phases with more than 50 winner's moves were deemed drawn.[2] Looking at the effect of giving the defender the null move has been proposed [9] and similarly, one might consider the effect of removing the ability either to capture or to mate in the current phase of play.[3]

The proposal here is to address the study community's *effective uniqueness* question algorithmically using EGTs. This is to be done by defining appropriate chess variants *Chess(SP)*: *SP* is a set of positions, each won for White in Chess but defined to be a draw in Chess(SP). The impact of these changes on the values of positions in another set *TP* of White wins is a measure of the criticality or importance of the positions in *SP* to those in *TP*. It may be determined from the difference, $\Delta(EGT, EGT_{SP})$, between Chess' EGT and Chess(SP)'s EGT.

Section 2 defines a set of scenarios where the questions about position criticality may be addressed using the Chess(SP) approach: Section 3 defines the response to each scenario. Section 4 details the algorithm, considers available efficiencies, and estimates the workload in the context of Chess' Win Studies [13]. Section 5 reviews the first implementation and production use of the approach by the second author in the game of Starchess.

Table 1. The cited exemplar positions

Id	Date	HHdbIV	Force	Position	move	Val.	DTM	Notes
UP1	2000	---	KQQKQP	wKg4,Qg8,h8/bKf1,Qb4,Pd2	w	1-0	60	Karrer: DTM=20 if d1=N is not possible
UP2	2009	75917	KQPKQ	wKc8,Qf4,Pg7/bKh5,Qh1	w	1-0	13	Konoval and Bourzutschky: P=R and =N
UP3	2010	---	KRRPKQ	wKc2,Rb5,d5,Pg5/bKh5,Qf8	w	1-0	36	Konoval and Bourzutschky: P=R, =B and =N
S1	1895	3477	KPKR	wKb6,Pc6/bKa1,Rd5	w 1	1-0	26	Saavedra and Barbier: most documented study
S2	2009	75649	KPPPKPPP	wKf1,Pa4,d5,g5/bKh8,Pa5,d6,g7	w 1	1-0	83	Hornecker study
S2'	2009	75649	KQPKQP	wKc6,Qf1,Pd5/bKg5,Qd4,Pd6	w 17	1-0	64	Hornecker study: sideline after 16. Qf1+
S3	1924	9797	KRNKNN	wKc6,Ne5,Rg5/bKd8,Nf8,h6	w 1	1-0	26	Rinck study: 7 winning moves at pos. 1w
S4	1924	9686	KNPKPP	wKf1,Ne2,Pg2/bKe3,Pf4,g3	w	1-0	36	Reti and Mandler study; p3w is a B1-M zug
PH	1777	956	KQKR	wKc6,Qa5/bKb8,Rb7	w	1-0	10	Philidor: B1 zug
KH	1851	1822	KBBKN	wKd5,Ba4,f8/bKb6,Nb7	w	1-0	57	Pseudo-fortress long thought to be drawn
B1Z	2011	---	KPPKPP	wKg5,Pe6,f7/bKg7,Pe7,g6	w	1-0	18	Elkies: a *vital* B1 zug needing B1Z/btm to win
B1Z'	2011	---	KPPKP	wKg5,Pe6,f7/bKg7,Pe7	w	1-0	13	Elkies: not a vital B1 zug

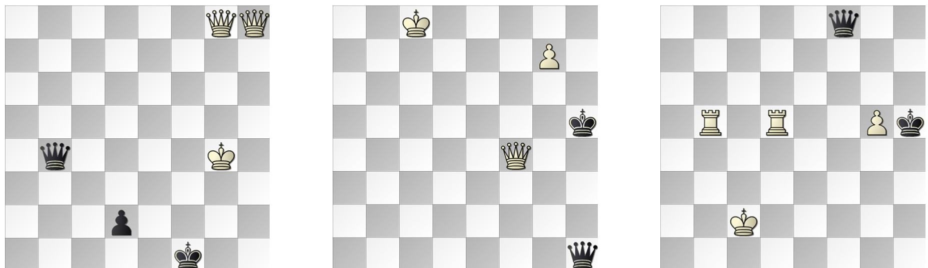


Fig. 1. Wins *UP1-3* requiring underpromotions, found using Chess variant EGTs

[2] 50.15% of wtm and 70.98% of btm wins in KBBKNN are '50-move draws' [11].

[3] i.e., before the next Pawn-push, capture and/or mate, when the move-count is zeroed.

2 Scenarios and Questions to Be Considered

The scenarios are constrained to that part of Chess for which the perfect information of EGTs is available. As is the convention in Chess Win Studies, White has a win throughout the line $\{P_1, \ldots, P_n\}$ and plays move $m_c{:}P_c{\rightarrow}P_{c+1}$ from the 'current' position. However, there may be alternative value-preserving moves $m_{c,j}{:}P_c{\rightarrow}P_{c,j}$. The moves which are suboptimal in any available metric[4] are $m_{c,1}$ to $m_{c,j1}$ while those optimal in some available metric are $m_{c,j1+1}$ to $m_{c,j2}$. *SP* and *TP* are two sets of positions theoretically won for White. *Chess(SP)* is a variant of chess only in that the positions in *SP* are deemed to be drawn, perhaps creating further draws in Chess(SP).[5] The question is 'What are *TP's* positions' values in Chess(SP)?' This notation is used:

S_i ≡ a reference to Table 1, $^!$ ≡ a move which seems clearly the best,
→ ≡ a move, *pn(w/b)* ≡ (wtm/btm) as at position *n*, *tw* ≡ time-wasting move,
' and " ≡ DT*x*-optimal, DT*x* being DTM here,
$^{(\prime)}$ ≡ all DT*x*-suboptimal moves are time-wasting moves,
"' ≡ the unique value-preserving move, ° ≡ the only move available,
(v)z ≡ (vital) zugzwang, *(±n)* ≡ a concession of *n* moves in DT*x* terms,
○ ≡ White to move (wtm) position, ● ≡ Black to move (btm) position.

2.1 The Main Scenario: The Win Study

In the main scenario, a Win Study challenges White to win. At position P_c, White plays move m_c but the dual winning moves $m_{c,j}{:}P_c{\rightarrow}P_{c,j}$ are also available. The study community's Grand Challenge question then, as discussed, is 'to what extent is move m_c unique: how significant are the dual moves?' Metric suboptimal moves $m_{c,j}$, $j=1,j1$ which allow Black to force White's win either to return to one of P_1-P_c or to arrive at P_{c+1} more slowly are *time wasters* and clearly inferior to a move from P_c which actually makes progress. The technical challenge addressed here is to discover which moves can be classified as time wasters, as a prelude to re-evaluating the essential uniqueness of the move m_c. HHDBIV has some 70,000 such s7m scenarios.

In many cases, White can *switchback*, retracting and repeating its last move, thereby wasting four plies. Move $m_{c,j}$ may simply be to some previous mainline position P_i, $i < c$, or to a position one Black move off the played line. However, $m_{c,j}$ can be the start of a large tree of alternative lines leaving the reader asking whether all options have been considered, given that neither side has to play metric-optimally. Larger move trees are less comprehensible, less easily verified, and may even fall short of a complete proof that a move is merely wasting time.

Some examples show that a generally-applicable method is required to address all situations and, regardless of their complexity, to produce uniformly and easily comprehensible, verifiable proofs about time-wasting moves. Studies *S1-S4*, see Figures 2-5, will suffice to indicate the main issues and open-ended range of complexity.

[4] e.g., DTC ≡ Depth to Conversion (of force), DTZ ≡ Depth to (move-count) Zeroing move.
[5] As is easily proved, existing draws or Black wins in Chess remain so in Chess(SP).

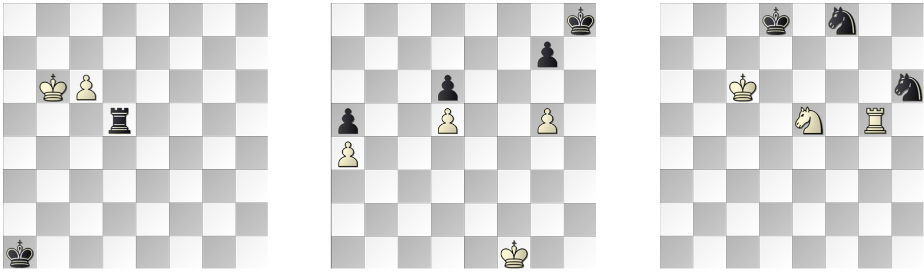


Fig. 2. Studies with ignorable time-wasting moves: *S1* (Saavedra and Barbier), *S2* (Hornecker) and *S3* (Rinck)

The 1895 study *S1* by Saavedra and Barbier is justly famous for its underpromotion and colorful history [8]: its economy and brevity are also laudable. The solution is **1. c7'''** *p1b* **Rd6+' 2. Kb5''' Rd5+''** *p3w* **3. Kb4''(') Rd4+ 4. Kb3'** *p4b* **Rd3+'' 5. Kc2''' Rd4!** inviting the instinctive 6. c8=Q which only draws after 6. ... Rc4+''' 7. Qxc4''' stalemate. **6. c8=R''** ignores White's seductive Queen by the board. After **6. … Ra4', 7. Kb3'''** wins by threatening both Rook and King.

However, White has alternative wins, see Figure 3, at moves 3, 4, and 6 which potentially undermine the uniqueness of the solution. 3. Kb6 and 6. Kb3 regress immediately to respectively *p1b* and *p4b*. 4. Kb5 allows 4. ... Rd5+'' *p3w*. The increased depth of win shows that two moves have been wasted in each case. The time-wasting is easy to see as the cycle is completed with at most one line and one sideline move: the solution [13] does not even acknowledge these moves. If clearly inferior moves invalidated studies on technicalities, much would be lost and the delights of such as the study by Saavedra and Barbier would be denied to a potential audience.

White also has the *dual* 4. Kc3' Rd1'' 5.Kc2''' Rd4 *p6w* so it is clear that Black can force White back to the mainline downstream. A further question then about dual moves is whether they allow Black to force White's win back to the mainline and how quickly this can be done.

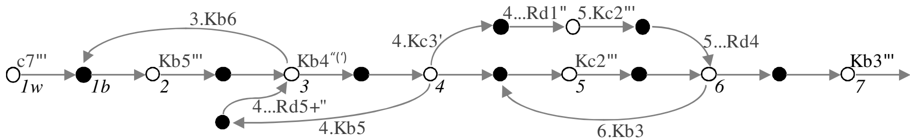


Fig. 3. A graph of the Saavedra study

Rusz [25] recently described his more challenging demonstration of a time-wasting move which rescued Hornecker's 2009 study *S2*. The solution is **1. g6! Kg8° 2. Ke2! Kf8! 3. Kd3! Ke7! 4. Kc4! Kf6! 5. Kb5! Kxg6! 6. Kxa5!** {KPPKPP, DTM = -77} **Kf5'' 7. Kb5''' g5'' 8. a5''' g4'' 9. a6''' g3'' 10. a7''' g2'' 11. a8=Q''' g1=Q'' 12. Qa3''' Ke4'' 13. Kc6''' Qd4''** *p14w* **14. Qa5''' Kf5''** *p15w* **15. Qb5''(')** (15. Qa3 *tw* Ke4'' 16. Qa5 ''' *p14b*) **15. ... Kf6** *p16w* **16. Qb6''(')** *p16b* **Qxb6+ 17. Kxb6''' Kf5' 18.Kc7''' Ke5' 19. Kc6'''**, a type A3 Trébuchet zugzwang or *zug* lost for Black.

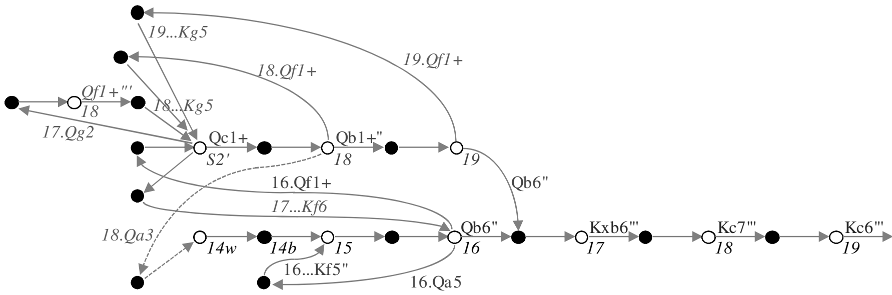


Fig. 4. A graph of the Hornecker study from position *p14w*

Position *p16w* is the focus: here, 20 sideline moves are required to show that White's moves 16. Qa5 and 16.Qf1+ are no more than time-wasters. 16. Qf1+ is ultimately shown to progress no further than the mainline move 16.Qb6"[(i)].

16. Qa5 another example of a *switchback*, move-reversing move 16...Kf5" *p15w*.
16. Qf1+ Kg5 q.v. *S2'* 17. Qc1+
(17.Qb5" reversing move 16w 17...Kf6 p16w; 17.Qg2+ Kf6" 18.Qf1+''' Kg5 S2')
17...Kf5 18. Qb1+"
(18.Qf1+ Kg5 S2'; 18.Qa3 Ke4" p14w) 18...Kf6 19.Qb6" *p16b (19.Qf1+ Kg5 S2').*

Although White has had no more than three winning options at any time, it is becoming clear that chess annotation and graphs reflect rather than reduce the complexity of proofs that moves are mere time-wasters. They redundantly detail White's unavailing attempts to make alternative progress rather than just stating that this is impossible. Further, manual proofs may not be complete and correct.

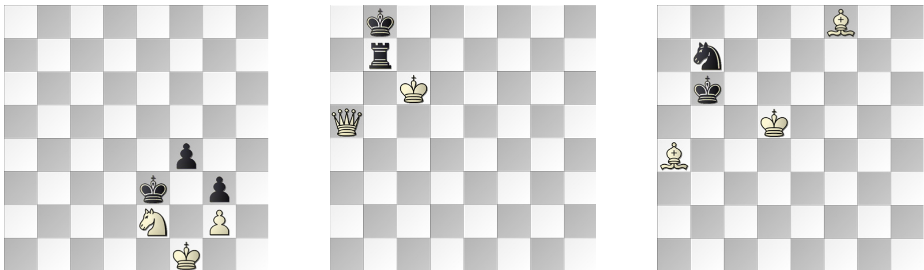


Fig. 5. S4 (Reti and Mandler), *PH* (Philidor) and *KH* (Kling & Horwitz)

The Rinck KRNKNN study *S3* of 1924 presents an even greater challenge which is not accepted here. Rinck's solution is **1. Rh5" Ng8" 2. Rh8" Ne7+" 3. Kb5" Ke8" 4. Kc5" Nf5"/Nc8 5. Ng6" Kf7!" 6. Nxf8''' Kg7 7. Rh5/Rh1'**. However 1. Rg7, Kd5, Rg1, Rg2, Nc4, and Rg3 also win.[6] It is clear that these moves are not what Rinck had in mind but unclear which of them if any allow Black to force a return to the initial position. White has alternatives for all moves except the key 6. Nxf8'''.

[6] Conceding 30, 39, 88, 101, 132 and 170 moves in DTM terms, and 32, 41, 90, 103, 133 and 170 moves in DTC/DTZ terms.

It is equally impractical to show via explored lines that the alternative moves in the 1924 study *S4* by Reti and Mandler [1] are time-wasters. The solution begins **1. Ng1''' Kd2" 2. Nf3+" Kd3"**, a type B1-M zug [2,9], that is, a won position that would be won more quickly in DTM terms if White could *pass*. White aims to return to this physical position but with Black to move. The solution continues **3. Ke1" Ke3" 4. Ne5" Ke4" 5. Nc4" Kd3" 6. Nd2" Ke3" 7. Nf3" Kd3" 8. Kf1"**. At position *p2w*, 2. Ne2 is an obvious time-waster (2. … Ke3" *p1w*) but 2. Nh3 is less easily discounted. The initial depth-concession is again only two moves but the Knight may explore the board further; similar opportunities are on offer down the main line. The analysis of these explorations in the study solution would scarcely be enlightening and the proof of time-wasting has to be in higher-order chessic terms. It would be better to use mathematical rather than chessic logic to show that 2. Nh3 is merely a time-wasting move. White wins after **8. … Ke3" 9. Ne1" Kd2" 10. Nc2" Kd1 11. Nb4" Kd2" 12. Nd5"**.

2.2 Supplementary Scenarios

In scenario 2, White is to move from a type B1-*x* zug. Three examples are positions *B1Z*, *B1Z'*, and the Reti-Mandler study's *p3w* as noted above. This suggests this question 'Is the B1-*x* zug a *vital* B1 (*VB1*) zug, that is one from which White's win can be forced to include the btm side of the zug?' There are many more B1-M zugs in chess than value-critical zugs, and over 6,000 B1-M zugs in HHDBIV [9].

Scenario 3 sees White, a computer program, winning a game but seeking to avoid a 50-move draw-claim [7, Article 9.3] from a fallible opponent as DTR > 50 [12].[7] White's strategy is to avoid repetition[8] [7, Article 5.2d] by considering past positions to be 'drawn', temporize and hope the opponent reduces DTR.

Scenario 4 focuses on positions highlighted as significant in endgame theory. Examples, see Figure 5, include Philidor's KQKR position *PH* [20, Ch.3] and the Kling and Horwitz pseudo-fortress position *KH* [19, Ch.5]. The question here is 'What related positions are not wins if these positions are not wins for White?'

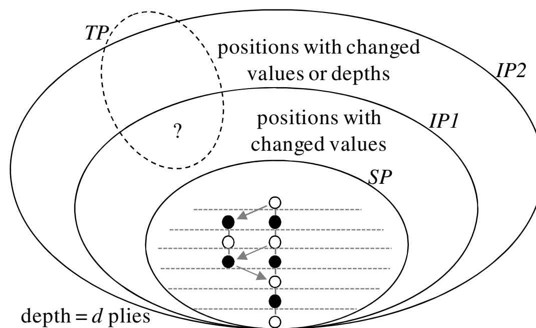


Fig. 6. Sets *IP1* and *IP2* show the impact of set *SP*, particularly on *TP's* positions

[7] DTR ≡ Depth by the (draw-claim) rule: DTR ≡ smallest *k* giving a win under a *k*-move-rule.

[8] Historic Dominguez-Perez/Polgar (World Cup, 2011): the unnoticed *p95b=p105b=p107b*! Rusz asks if J. Polgar can win without revisiting *p105b* after regressing with 105…Bf5.

3 The Algorithm: A Generic Response to the Scenarios

The thematic question of the scenarios is 'Given that the set *SP* of white wins are defined to be draws, which of the White wins in set *TP* become draws?' Figure 6 shows the set *SP*, *TP* and the 'upstream' sets *IP1* and *IP2* of positions of which the theoretical values or DT*x* depths are different in Chess(SP). The generic algorithmic response then is to:

- define the set *SP* of White-win positions, and thus the variant game *Chess(SP)*,
- define the set *TP* of White-win positions whose Chess(SP) values are sought,
- compute the relevant EGT_{SP} Chess(SP) EGT until set *TP* is accounted for,
  Chess(SP) like Chess has a lattice of endgame phases;
  therefore, its EGTs are computable as are those of Chess,
- examine, for positions in set *TP*, the EGT-difference $\Delta(EGT, EGT_{SP})$
  the differences are caused only by *SP's* positions being draws not wins,
  $P \in TP \cap IP1 \Leftrightarrow P \in TP$ is a win in Chess but a draw in Chess(SP).
  In more efficient codes, this may be seen during the computation.
  This allows the computation to be aborted without generating EGT_{SP} fully.

In scenario 1 at position P_c, *SP* is a combination of $\{P_1, \ldots, P_{c-1}\}$, $\{P_c\}$ and $\{P_{c+1}\}$; *TP* $\equiv\{P_{c,j}\}$, the set of P_c*'s* successor positions, other than P_{c+1}, won for White. If and only if $P_{c,j}$ is in set *IP1*, i.e., a Chess(SP) draw, the win in Chess from $P_{i,j}$ can be forced to pass through a position in *SP*. This means move $m_{c,j}$ is a time-waster, the precise reason being determined by the Chess(SP) drawing line(s) from $P_{c,j}$.

In scenario 2, *Pw* is a wtm type B1-*x* zug and *Pb* is its btm equivalent; $SP \equiv \{Pb\}$ and $TP \equiv \{Pw\}$. *Pb* is essential to White's win from *Pw* in Chess if and only if *Pw* is a draw in Chess(SP).

In scenario 3, White has a win, may play suboptimally with regard to all metrics, but does not wish to repeat position. $SP \equiv \{P_1, \ldots, P_c\}$. Time-wasting moves to positions in *IP1* are avoided as are overlong phase continuations.

In scenario 4, *SP* ≡ {*significant position*}, e.g., position *PH* or *KH*, is considered a near-refuge for Black. By making it an actual drawing sanctuary in Chess(SP), it is possible to assess its importance to White's winning chances in Chess.

Other scenarios involve deep wins or downstream-convergence in Win Studies, Draw Studies and value-critical zugzwangs. For clarity, these are not included in this first exposition of the Chess(SP) approach.

Table 2. HHdbIV Endgame study s7m mainline positions to be evaluated[9]

	wtm positions with alternative moves		EGT creation at 'Konoval tempo' (hrs)	
n	**#pos, *n* men**	**#pos., *n* pieces**	**DTC/M EGTs**	**DTZ EGTs**
2	0	5,057	0.00E+00	3.22E-03
3	299	9,911	1.22E-02	4.03E-01
4	5,868	28,890	1.53E+01	7.52E+01
5	34,401	21,864	5.73E+03	3.64E+03
6	30,231	5,077	3.22E+05	5.42E+04
Totals	70,799	70,799	3.28E+05	5.79E+04

9 Chess *men* are *pieces* or *Pawns*: the pawns need not be moved in creating the EGT_{SP}.

4 Generating *Chess(SP)* EGTs: Examples and Efficiencies

Figure 3 and a 'manual' implementation of the Chess(SP) algorithm for the study *S1*, position *3w*, indicate the rapidity of the 'Chess(SP) test'. The question is whether move 3. Kb6 is a time-waster or not. $SP \equiv$ {pos. *3w*} and $TP \equiv$ {pos. *1b*}. Mainline positions *2b*, *2w* and *1b* immediately revert to *draw*: Black takes any drawing option and White is denied its only winning move. However, position *1b* was in the mainline of the study anyway so the generation of EGT_{SP} was unnecessary. Showing that 4. Kb5 is a time-waster requires the creation of the full EGT_{SP} if $SP \equiv$ {pos. *4w*}. However, this is rapidly obvious if $SP \equiv$ {positions *1w-3b*} when '*3b* draw' implies the position after 4. Kb5 is a draw. The reader may care to show by similar means that 6. Kb3 is a time-waster and that the line starting 4. Kc3 can be forced to position *6w*.

If thousands of EGT_{SP} designer-EGTs are to be created, it is appropriate to consider the work involved and how it might be reduced. This will depend on which EGT-generator is evolved to create EGTs for Chess(SP). For example, Nalimov's code [18] is slower than Konoval's single-threading code which computed the KQBNKQB EGT in 3.5 weeks [27] and can compute the KQPKQ EGT in 10 minutes.

There are efficiencies which apply to creating any EGT and efficiencies which are specific to generating EGTs for Chess(SP). Konoval has used PENTIUM Assembler in the inner loops of his program and a relatively simple position-indexing scheme which facilitates the fully retrograde production of EGTs.[10]

EGT generation may be speeded up considerably if a trusted EGT for the same endgame is already available, especially if this is a WDL EGT.[11] It is unnecessary to evaluate a position expensively as a potential loss if it is already known to be a draw or a win. This economy is in principle available when creating EGT_{SP}.

When generating EGT_{SP}, it is worth noting that:

- Chess draws and Black wins are unaffected by deeming *SP's* positions drawn,
- 'downstream' $Q \notin SP$ with $DTx(Q) \leq minDTx(P \in SP)$ are also unaffected,
- the EGT_{SP} need only be for positions with Pawns in relevant positions,
- a WDL EGT_{SP} is sufficient to determine set *IP1*,
- the creation of EGT_{SP} may be halted when set *IP1* has clearly been identified or the Chess(SP) values of the positions in set *TP* are known,
- iterative steps in creating EGT_{SP} can identify draws as well as wins,
- a larger *SP* does not slow the evaluation of *TP's* positions in Chess(SP),
- if $SP2 \supset SP1$, e.g. $\{P_1 \ldots P_c\}$ and $\{P_c\}$, EGT_{SP2} may be derived from EGT_{SP1}.

Based on 'Konoval performance', Table 2 estimates the computer time needed to identify systematically all time-wasting moves from the s7m mainline positions in HHDBIV's studies. The estimates do not include any Chess(SP)-specific efficiencies even though 50-fold efficiencies have been seen in Rusz' production work below. Given a suitable infrastructure to manage thousands of independent tasks, the elapsed time may be greatly reduced by the use of multi-core computers, networks of computers and crowd-sourcing. Parallelism is also possible within the set-manipulating EGT-generation algorithm for the largest EGTs for Chess(SP).

---

[10] i.e., unmoving from positions of depth *d* plies to discover positions of depth *d+1* plies.

[11] WDL EGTs merely hold 2-bit information about wins, draws and losses but not about depths.

5 The First Implementation of the Algorithm: Starchess

As this proposal, to generate EGTs for the chess variant *Chess(SP)*, has not been widely promulgated, no existing generator of Chess EGTs has yet been generalized to do so. However, the second author, a leading authority and world champion in the game Starchess [26], has generalized his EGT generator. Starchess was invented by László Polgár in 2002 and it is only necessary here to mention the star-shaped board of just 37 hexagons[12], the Knight's move, e.g., 19-2/3/9/16, and the humbled Rook which can only move vertically. There are many short, combative Starchess games: openings tend to be more tactical than in Chess but endgames are of similar length.

After generating the sub-6-man (s6m) Starchess EGTs, Rusz identified 9,967,573 type B1-M zugs. As in Chess itself, B1-M zugs with *zug depth*[13] $zd < 3$ predominate: 9,852,307 have $zd=1$ and many merely call for a waiting move; 78,001 have $zd=2$.[14] As in scenario 2, the question then arises as to which are *vital* type B1 zugs. A motivation is that if White can be forced to visit the btm side of the B1 zug in *tl* (transit

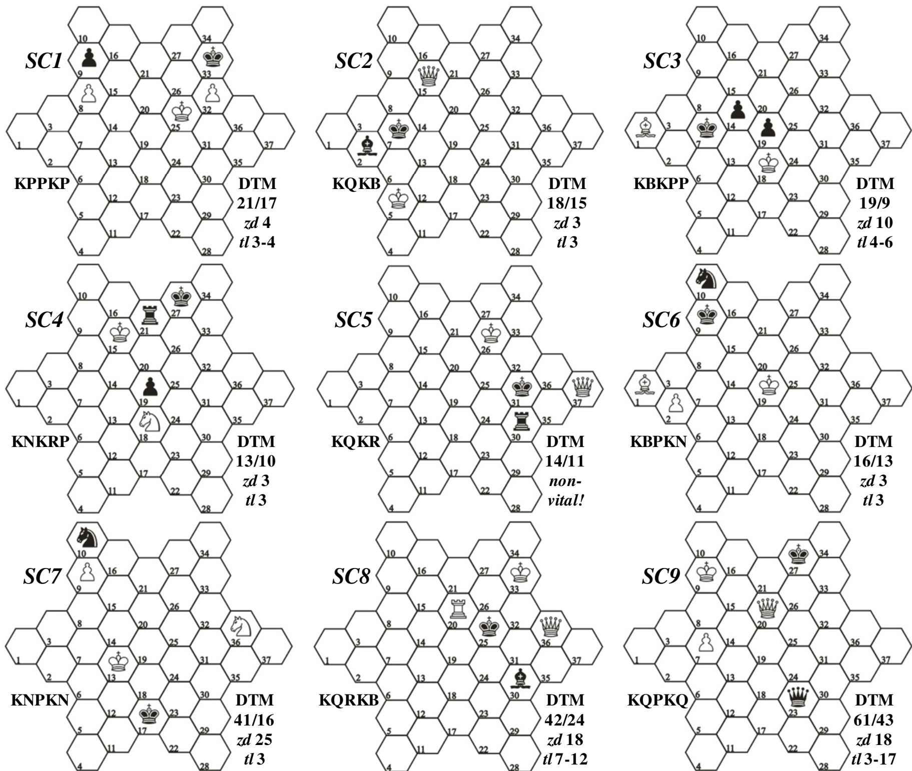


Fig. 7. Sub-6-man Starchess positions *SC1-9* on the Vital B1 zug theme

[12] The hexagons are numbered in columns, left to right, and bottom to top: '37' is on the right.

[13] The zug-depth *zd* of a B1-M zug is the difference between the wtm and btm DTM depths.

[14] Of 1,626,168,997 s6m positions, 0.61% are B1-M zugs: 0.0023% also have $zd > 2$.

length) moves, it is likely that a study-like scenario will be found. The technique, as in Section 3, is to see if, with the btm side of the B1 zug *drawn*, the B1 zug itself becomes a draw. If and only if it does, Black can force White's win to visit the btm position. The forced transits, wtm to btm position, may have different lengths.

Two observations speeded the identification of the vital s6m B1-M zugs, the first by Rusz and Starchess-specific. Vital B1 zugs in pawnful positions must have *zd*>2: both sides must play at least two moves in going from the wtm to the btm position as there are no moves which preserve symmetry across the single vertical axis of symmetry. This meant that 7,168,489 B1-M zugs could be ignored, a splendid economy. Secondly, as in any game, if the B1-M zug in set *TP* is seen to be a win despite the btm position being defined as drawn, the generation of the Starchess(SP) EGT may be discontinued. Thus, over 50 B1-M zugs were examined in the time taken to generate an EGT. Fourteen of the 910 VB1 zugs found (30 4-man, 128 5-man pawnless) and one instructive non-VB1 zug, *SC5*, feature in Figures 7-8 and here.

KPPKP VB1 zug *SC1* is analogous to Lasker's chess study[15] HHDBIV#14482 [1, 23-25]. The *max_tl* line **1. K31'''** (1. K20? K32''' 2. K21' K25''' 3. K16' K20' 4. K9' K14''' *z* =) **1. … K27' 2. K24"(') K34" 3. K19"(') K33" 4. K25'''** *z p1* K34" 5. K20"(') K27" 6. P33+"(') K33' 7. K21''' K32' 8. K16''' K26" 9. K9''' K21" 10. K10° K20' 11. K16"(') 1-0. A marginally quicker force with *min_tl*=3 is **2. … K33 3. K25"(')** *z p1*.

KQKB VB1 zug *SC2* illustrates an important Q-triangulation. **1. Q20+''' K3" 2. Q21+"(') K7° 3. Q15"(')** *z p1* 1-0.

KBKPP VB1 zug *SC3*, *max_tl*=6**: 1. B17"(') K3' 2. B37"(') K2" 3. B26"(') K7" 4. B33" K8" 5. B27+"(') K7** *(-4)* **6. B1"(')** *z p1*. Black may also play the *min_tl*=4 line **2. ... K8** *(-3)* **3. B21+"(') K7** *(-3)* **4. B1"(')** illustrating the 'B-diamond' manoeuvre.

KNKRP VB1 zug *SC4*, *tl=3*, three VB1 zugs: *vz1* **1. K9"(') K33' 2. K16"(') K27' 3. K15"(')** *z p1* **K33' 4. K21"(') K34°** *vz2*, 9/4, *zd*=*tl*=5 **5. K20"(') K33"** *vz3*, 8/5, *zd*=*tl*=3 **6. N15+"(') K27" 7. N2"(') K33" 8. N18"(')** *z p6* N-triangle **K34" 9. K21"(')** *z p5* K33° 10. N15+"(') K34° 11. K26"(') P18° 12. N33''' P17=(Q/R/B/N)° 13. N16#''' 1-0.

KQKR position *SC5* just fails to be a Vital B1 zug**.** The wK triangulates from/to the zug: **1. K33" K24" 2. K27" K31' 3. K26"** *z p1*. However, setting *p3b* to *draw* neatly reveals the dual win from *p1w* (annotation in *Chess(SP)* terms): **1. K21"(') R28" 2. K15"(')** distant wK/bR opposition! **2. … R29" 3. K20' R30" 4. K14"(') R28" 5. K19"(') R30" 6. K13"(') R28" 7. K12"(') R30" 8. K18"(') R29" 9. K17"(') R30" 10. K22"(')**.

KBPKN VB1 zug *SC6*, *tl*=3**:** *vz1* 1. **B34''' K16"** *vz2*, 15/12, *zd*=3 **2. B8"(') K9" 3.B1"(')** *z p1* (3.P3? K16''' *z*) 3. ... K16" 4. B34"(') K9' 5. P3"(') *z* K16" 6. B8''' *z* K9" 7. K25"(') *z* K16" 8. K26"(') *z* K9" 9. K21"(') 1-0.

KNPKN VB1 zug *SC7*, *tl=3***:** one of three positions with a record *zd=25.* **1. K19''' N3" 2. K14"(') N10** *(-22)* **3. K13"(')** *z p1*. If **2. … K18"** then **3. N20''' N19" 4. P10N'''** reaches an interesting KNNKN endgame, a general win.

KQRKB VB1 zug *SC8*, *max_tl*=12, the second-longest known transit: **1. Q26+"(') K31° 2. R21"(') B24" 3. Q20+"(') K36" 4. Q25"(') B31" 5. Q23"(') B35" 6. Q26+"(') K31° 7. Q32+"(') K24° 8. Q19+"(') K23" 9. Q17+''' K24" 10. Q18+"(') K25** *(-6)* **11. Q36"(') B30" 12. R20"(')** *z p1* 1-0. A *min_tl*=7 line diverges **2. … B35** *(-4)* **3.Q32+"(') K24° 4.Q19+"(') K31** *(-1)* **5.Q18+"(') K25** *(-6)* **6.Q36"(') B30" 7.R20"(')** *z p1*.

---

15 wKc5,Pa5,c6/bKc7,Pa6: 1. Kd5''' Kc8" 2. Kd4' Kd8' 3. Kc4"(') Kc7 *(-1)* 4. Kc5"(') *z p1*.

KQPKQ VB1 zug *SC9*, *max_tl*=17**: 1. Q15''(') K33'' 2. Q14+''(') K34'' 3. K8''(') K27'' 4. Q19''(') K26'' 5. K3''(') K27'' 6. Q20''(') Q29'' 7. Q17''(') K26''** *vz2* 54/50, *zd*=4, *tl*=4 **8. K2''(') Q34'' 9. Q19''' Q28+'' 10. K3''' Q29'' 11. Q17''(')** *z p8* **K27'' 12. K8''(') Q31'' 13. K14''(') Q24+'' 14. K15''' Q25+' 15. Q20''(') Q13+** *(-1)* the only suboptimal move in the line (instead of 15. ... Q24'') **16. K8''(') Q23'' 17. K9''(')** *z p1* 1-0. There is a *min_tl*=3 line diverging **2. … K27** *(-15)* **3.Q20''(')** *z p1* 1-0: *max_tl* – *min_tl* = 14.

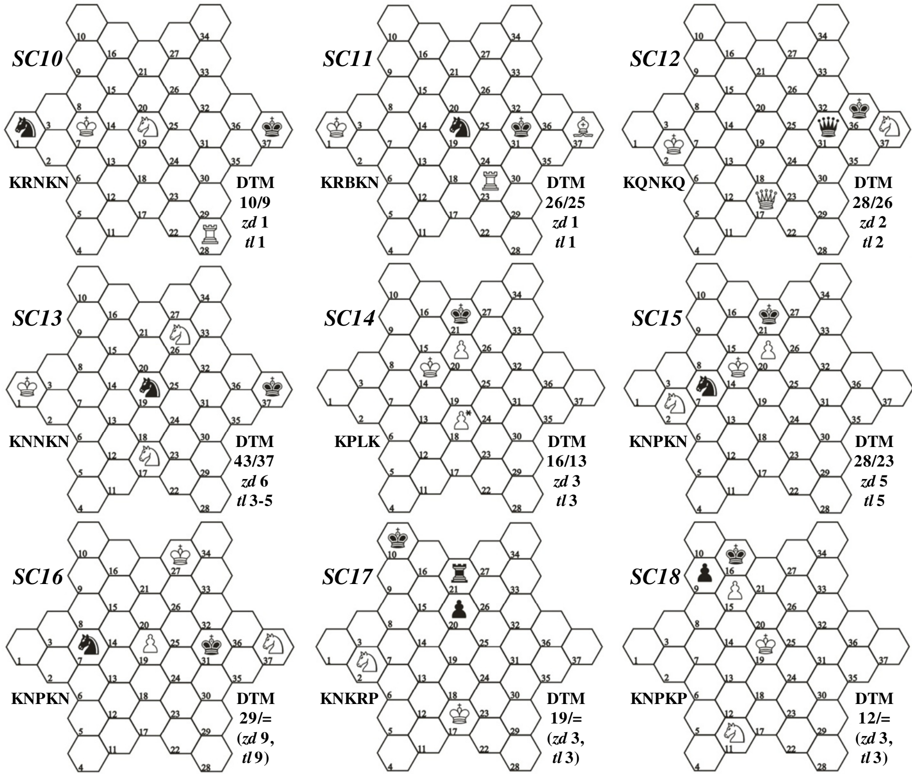


Fig. 8. The six s6m reflected VB1 zugs and three studies featuring VB1 zugs

The VB1 zugs of Figure 8 are the six s6m *reflected* ones: *SC10-13* are reflected in the horizontal axis and *SC14-15* in the vertical axis. *SC10-11* have *zd*=1 and *SC12* uniquely has *zd*=2. From *SC13*, Black has the choice to *reflect* the vital zug or not. *SC14* is the only reflected VB1 zug featuring a *Limping Pawn*, a pawn which appears to be able to move two squares but in fact cannot as it has already captured a man. The opening position and the dance of the Knights makes *SC15* visually remarkable.

KRNKN VB1 zug *SC10*, *tl*=1: **1. R34'''** *z p1*-reflected (denoted *p1-r)* 1-0.
KRBKN VB1 zug *SC11*, *tl*=1: **1. R26'''** *z p1-r* 1-0.
KQNKQ VB1 zug *SC12*, *tl=2*: **1. Q21+''(') K35'' 2. K3''(')** *z p1-r* 1-0.
KNNKN VB1 zug *SC13*, *max_tl*=5**: 1. N31''(') N35** *(-1)* **2. N26-13''(') N19'' 3. N14''(')**. Now Black has **3…K36' 4. N26''(') K37'' 5. N17''(')** *z p1* or **3…K35' 4. N23''(') K37'' 5. N21''(')** *z p1-r*. *min_tl*=3: **1. … K36** *(-3)* **2.N14''(') K37'' 3.N17''(')** *z p1* 1-0.

KPLK VB1 *SC14*, *tl*=3: **1. K19''' K27'' 2. K24''(') K21'' 3. K25'''** *z p1-r* 1-0. White has the option, with the same tempo, not to reflect this VB1 zug. If the Limping Pawn were not limping, the position would be a non-vital type B1 zug (DTM 14/13): 1. K19'' K16' 2. P21=Q' K21 *z* 3. K14' K26 *z* 4. K15''(') K25 5. P20''(') 1-0.

KNPKN VB1 *SC15*, *tl=5*: **1. N9''' N17'' 2. K19''' N31'' 3. N26''(') N27' 4. K25''' N31'' 5. N35''(')** *z p1-r* 1-0.

The identification of the 910 s6m Vital B1 zugs has revealed about 250 study scenarios and many positions of pedagogic value. Here are three of them.

SC16, with a positional draw in line A and Knight sacrifices in both lines: 1. **P20'''**

a) **1…K30 2. N25'''!** (2. K26? K23''' 3. N25 N24''' 4. K27 K22''' 5. N37 N7''' 6. K26 K23''' 7. N25 N24''' positional draw) **2. …N24 3. N16'' K23'' 4. N19'' K18 5. N11''!**, or

b) **1…K32''** *vz zd=tl=9* **2. N29'' N24'' 3. N19''(') K25'' 4. N36''(') N7'' 5. N34''' K32'' 6. N16'''! N24'' 7. N3''(') K25' 8. N12''' K32'' 9. N29''(') N7'' 10. N37''(')!** *p2* **K31'' 11. K26''(') K30'' 12. N33''(') K23'' 13. N19''(') K17''** Black defends against the N11! sacrifice **14. N29''' N24 15. N37''! N7 16. N25''** 1-0.

SC17: 1. **N18'''!** (1.N19?) **1…P19' 2. K23''' K16' 3. K30''' R20' 4. K31''' K21 5. K25'''**

a) **5…K27' 6. K20''' K33''** *vz zd=tl*=3 **7. N15''(') K27'' 8. N2''(')! K33'' 9. N18''(')** *p7w* **K34'' 10. K21''(') K33° 11. N15''(') K34° 12. K26''(') P18° 13. N33''' P17=Q' 14. N16'''#.**

b) **5…K16' 6. K20''(') K9''** *vz-r zd=tl=3* **7. N26+''(') K16'' 8. N35''(')! K9'' 9. N18''(')** *p7w* **K10'' 10. K21''(') K9° 11. N26''(') K10° 12. K15''(') P18° 13. N9''' P=17Q' 14. N27'''#.** Two echo variations with N-triangulation.

SC18: **1. N7+'''** (1. K20? P15'''=) **1…K15''** *vz zd=tl=3* **2. N24''' K8'' 3. N11''(')! K15'' 4. N7''(')** *p2* **K8'' 5. N24''(') K3° 6. N11''(')!** an interesting, original N-manoeuvre **K8' 7. K20''(') K3° 8. K14''(') K1' 9. N19''(') P8+'' 10. K15'' P7° 11. K8'' P6° 12. N9''#.**

6 Summary

There is a clear need to identify all s7m time-wasting moves in the mainlines of studies in HHDBIV. This is a precursor to refining the study community's artistic judgment of that corpus [23] and would add considerable value to both. Proof statements should be derived by algorithm, reliable, economical, irredundant, comprehensible, and verifiable: this is not the case at present. The approach using *Chess(SP)* EGTs in principle yields proofs of moves' time-wasting status meeting the requirements stated. Further, it provides a tool to identify time-wasting moves in the future. This paper implicitly challenges the authors of EGT generator software to generalize that software to include variant games where some of White's won positions are deemed to be draws.

Similar questions of interest to endgame theoreticians, including those concerning lines from the deepest positions, zugzwang effects, 'downstream convergence' and Draw Studies, may be addressed using the same Chess(SP) approach. Other sources of endgame play, e.g., [19] [20], and studies' sidelines are worth inspection.

The authors particularly thank CESC [11], Rafael Andrist, John Beasley, Ian Bland, Eiko Bleicher, Marc Bourzutschky, Noam Elkies, Harold van der Heijden, Harm Műller, John Nunn, John Roycroft, John Tamplin, and the three anonymous referees for their interest in and contributions to this paper.

References

1. Beasley, J.: Depth and Beauty, the chess endgame studies of Artur Mandler (2003)
2. Bleicher, E., Haworth, G.McC: 6-Man Chess and Zugzwangs. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 123–135. Springer, Heidelberg (2010)
3. Bleicher, E.: DTM EGT query facility (2007), `http://www.k4it.de/index.php?topic=egtb&lang=en`
4. Bourzutschky, M., Konoval, Y.: News in Endgame Databases. EG 17-185, 220–229 (2011)
5. Bourzutschky, M.S., Tamplin, J.A., Haworth, G.McC: Chess endgames: 6-man data and strategy. Theoretical Computer Science 349(2), 140–157 (2005) ISSN 0304-3975
6. Fahrni, H.: Das Endspiel im Schach, Leipzig (1917)
7. FIDE: The Laws of Chess. FIDE Handbook E.1.01A (2009), `http://www.fide.com/component/-handbook/?id=124&view=article`
8. Grondijs, H.: No Rook Unturned. A Tour around the Saavedra Study (2004)
9. Haworth, G.McC, van der Heijden, H.M.J.F., Bleicher, E.: Zugzwangs in Chess Studies. ICGA Journal 34(2), 82–88 (2011)
10. Haworth, G.McC, Bleicher, E., van der Heijden, H.M.J.F.: Uniqueness in chess studies. ICGA Journal 34(1), 22–24 (2011)
11. Haworth, G.McC: The Scorched Earth Algorithm. Presentation to the UK Chess Endgame Study Circle. Pushkin House, London (2009)
12. Haworth, G.McC: Strategies for Constrained Optimisation. ICGA J. 23(1), 9–20 (2000)
13. van der Heijden, H.M.J.F.: HHdbIV, HvdH's Endgame Study Database IV (2010), `http://www.hhdbiv.nl/`
14. Karrer, P.: KQQKQP and KQPKQP. ICCA Journal 23(2), 75–84 (2000)
15. Konoval, Y., Bourzutschky, M.: First 6-man study requiring all three underpromotions. British Chess Magazine 130(1) (2010)
16. Konoval, Y., Bourzutschky, M.: Malyutka requiring P=R and P=N. 64 Shakhmatnoe Obozrenie 7 (2009)
17. Lasker, E.: Common Sense in Chess (1896)
18. Nalimov, E.V., Haworth, G.McC, Heinz, E.: Space-Efficient Indexing of Endgame Databases for Chess. In: van den Herik, H.J., Monien, B. (eds.) Advances in Computer Games 9, vol. 1, pp. 93–113. Institute for Knowledge and Agent Technology (IKAT), Maastricht (2001) ISBN 9-0621-6576-1
19. Nunn, J.: Secrets of Minor-Piece Endings. B.T. Batsford, London (1995)
20. Nunn, J.: Secrets of Pawnless Endings. B.T. Batsford, London. ISBN 0-7134-7508-0. Expanded Edition 2 including 6-man endgames (2002). Gambit. ISBN 1-9019-8365-X (1994)
21. Pritchard, D.B.: Popular Chess Variants. Batsford (2000) ISBN 978-0713485783
22. Pritchard, D.B.: The Classified Encyclopedia of Chess Variants. 2nd revised edition, edited and published by J.D.Beasley.
23. Roycroft, A.J.: Test tube chess: a comprehensive introduction to the endgame study (1972)
24. Roycroft, A.J.: Expert against the Oracle. In: Hayes, J.E., Michie, D., Richards, J. (eds.) Machine Intelligence 11, pp. 347–373 (1988)
25. Rusz, A.: A technical discussion of the Hornecker study HHdbIV#75649 (2011), `http://kirill-kryukov.com/chess/discussion-board-/viewtopic.php?f=6&t=6036#p64616`
26. Starchess rules (2011), `http://polgarstarchess.com/Rules.doc`
27. Vlasak, E.: Endgame database news (2005), `http://www.vlasak.biz/tablebase.htm`
28. Winter, E.: Chess Notes #5814 (2011), `http://www.chesshistory.com/winter/winter51.html`

Appendix: Further Details of Some Positions in Table 1

UP1, Karrer [7]. wKg4,Qg8,h8/bKf1,Qb4,Pd2 wtm:
This position is DTM=60 but a mate in 20 if Pawn-promotion is to Queen only.

1. Kf5''' Qc5+'' 2. Ke4''' Qe7+'' 3. Kf4'' Qb4+' [Qc7+, Qd6+]
4. Ke3'' Qc5+' [Qb6+, Qe7+] as d1=N is not available 5. Kxd2'' {DTM=15}

With the P=N option, a DTM-minimaxing line starts:

1. Kf5''' Qc5+'' 2. Ke4''' Qe7+'' 3. Kf4'' Qb4+' 4. Ke3'' d1=N+'' {KQQKQN}
5. Kf3''' Qa3+'' 6. Kf4''' Qe3+'' 7. Kf5''' Qd3+' 8. Ke5'' Qc3+'' 9. Kd5'' Qa5+''
10. Ke6' Qb6+'' 11. Ke5' Qc7+'' 12. Ke4'' Qc2+''.

UP2, Konoval and Bourzutschky [8-9], wKc8,Qf4,Pg7/bKh5,Qh1 wtm, requiring Pawn-conversion in different lines to respectively Queen, Rook and Knight:

1. Kc7''' Qh3'
(1. ... Qg1 2. Qf7+''' Kh6' 3. g8=N+''' Kg5'' 4. Qg7+'' Kf4' 5. Qxg1''')
(1. … (Qa8/Qd5)'' 2. Qh2+''' Kg6 3. Qg2+''' Qxg2'' 4. g8=Q+''')
2. Qe5''' Kh6 3. g8=R'''.

UP3, Konoval and Bourzutschky [8], [10]. wKc2,Rb5,d5,Pg5/bKh5,Qf8 wtm, the first s7m study synthesising all three Rook, Bishop and Knight underpromotions:

1. g6''' Kh6'' 2. g7''' Qf2+'' 3. Rd2''' (DTM = 34) and now
3. ... Qf3 4. g8=R''' Qc6+'' 5. Kd1''' Qh1'' 6. Ke2'' (6. Kc2 Qc6'' 7. Kd1''' Qh1''),
3. ... Qf4'' 4. g8=B''', or
3. ... Qf1 4. g8=N''' Kh7' 5. Rb7+''' Kh8'' 6. Ne7'''.

PH, the Philidor position (1777). wKc6,Qa5/bKb8,Rb7 wtm/btm, a B1 zug:

btm: **1. … Rb1'' 2. Qd8+' Ka7° 3. Qd4+'' (Ka8/b8)'' 4. Qh8+'' Ka7'' 5. Qh7+''**.
wtm: **1. Qe5+' Ka8' 2. Qa1+' Kb8'' 3. Qa5''** arriving at the btm line above.

KH, the Kling and Horwitz position (1851). wKd5,Ba4,f8/bKb6,Nb7 wtm.
This position was long thought to be drawn: in fact, DTC = 45m and DTM = 57m. White has to force Black from the pseudo-fortress [28] and prevent a similar pseudo-fortress being set up. This DTC-minimaxing line shows the difficulty involved.

1. Bb4'' Kc7' 2. Bd2' Kb6'' 3. Be3+' Kc7'' 4. Bf2' Nd8' 5. Kc4' Nb7''
6. Bg3+' Kb6'' 7. Kb4'' Nd8'' 8. Bf2+'' Kc7'' 9. Kb5'' Ne6'' 10. Bg3+' Kd7''
11. Bd1'' Nd4+'' 12. Kc5'' Nf5'' 13. Be5' Kd6'' 14. Bc3'' Ne3'' 15. Bf3'' Kf5''
16. Bc6'' Nf1' 17. Kd5'' Ng3'' 18. Bd7+' Kf4'' 19. Bd2+' Kf3° 20. Bh6' Nf1'
21. Kd4'' Ng3' 22. Bc6+' Kg4'' 23. Ke5'' Ne2' 24. Be3' Ng3' 25. Bc5' Nh5'
26. Bb6' Ng3'' 27. Ba4'' Nf1' 28. Bd1+'' Kg3'' 29. Kd4' Kf2' 30. Kd3+'' Kg3'
31. Ke2'' Nh2'' 32. Bc7+'' Kh3'' 33. Ba4' Ng4'' 34. Bc6'' Kh4'' 35. Kf3'' Kh5''
36. Kf4'' Nf6'' 37. Kf5'' Ng8'' 38. Bf3+'' Kh6'' 39. Bd6'' Kg7'' 40. Kg5'' Kf7''
41. Bd5+'' Kg7'' 42. Bc4' Kh7 43. Bf8'' Kh8'' 44. Bd3'' Ne7' 45. Bxe7''

B1Z, Elkies' pedagogic example (2011). wKg5,Pe6,f7/bKg7,Pe7,g6 wtm.
The wK cannot capture bPe7 first: with the bK on f8, the g-Pawn just advances. Therefore the triangulation is necessary and the win must visit *B1Z* with btm.

1. Kf4' Kf8'' 2. Kg4'' Kg7'' 3. Kg5''(') *B1Z-btm* **Kf8'' 4. Kh6''(') g5° 5. Kxg5''**
Kg7° the type B1 zug *B1Z'*: the win need not visit *B1Z'* with btm. **6. Kf5'' Kf8''.**

On Board-Filling Games with Random-Turn Order and Monte Carlo Perfectness

Ingo Althöfer

Fakultät für Mathematik und Informatik,
Friedrich-Schiller-Universität, 07737 Jena, Germany
ingo.althoefer@uni-jena.de

Abstract. In a game, pure Monte Carlo search with parameter T means that for each feasible move T random games are generated. The move with the best average score is played. We call a game "Monte Carlo perfect" when this straightforward procedure converges to perfect play for each position, when T goes to infinity. Many popular games like Go, Hex, and Amazons are NOT Monte Carlo perfect.

In this paper, two-player zero-sum games are investigated where the turn-order is random: always a fair coin flip decides which player acts in the next move. A whole class of such random-turn games is proven to be Monte Carlo perfect. The result and generalisations are discussed, with example games ranging from very abstract to very concrete.

1 Introduction

When speaking about Monte Carlo search in game trees, two different procedures have to be distinguished:

1. pure Monte Carlo and
2. tree search, based on Monte Carlo evaluations.

Pure Monte Carlo [7] with simulation parameter T means that for the current position all feasible moves are generated. For each such move, T random games are played to the end, and the average score over the T games is determined. The move with the best average score is played. Often this procedure leads to a reasonable move; but the move selected is not always an optimal one.

Figure 1 shows a typical game tree of depth 2 where pure Monte Carlo at the root almost surely leads to a wrong decision when the number of simulations is large. We call a game "Monte Carlo perfect" ("MC-perfect" for short) when pure Monte Carlo converges to perfect play for each position, when T goes to infinity.

Many popular games like Go, Hex, and Amazons are known to be NOT Monte Carlo perfect. See also the GoMoku example in the short note [6]. The wrong decisions of pure Monte Carlo can be avoided when Monte Carlo evaluation of

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 258–269, 2012.
© Springer-Verlag Berlin Heidelberg 2012

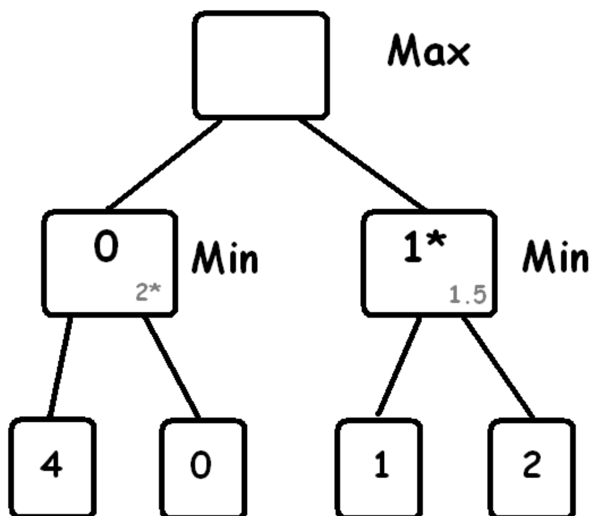


Fig. 1. A minimax tree of depth 2. Numbers in fat are the leaf values and true backed-up min values. Small numbers in grey are the Monte Carlo rates for the two nodes directly below the root. The stars mark where a perfect agent (fat star) and a pure Monte Carlo agent with large T (small grey star) would go, respectively.

positions is mixed with a step-by-step generation of the game tree (and minimaxing in some way or another). Seminal papers on such approaches are [8] and [11]. Only recently, a large class of games has been shown to be Monte Carlo perfect [12]: "selection games" with random turn-order where the player to move can fill any free cell with a piece of his own color and where the outcome depends only on the final full board. At the first glance this result looks much too general to be true, but it is. Its proof is beautifully simple and goes to the heart of the matter. Having proved, we can sit back and enjoy all its consequences, including the design of phantastic and unbelievably strange new games.

In this paper we generalize selection games in the following way. Our games are "board filling games" for two players A and B with zero-sum payoff. The board consists of n cells. In each move one free cell is filled either by "0" or "1". The game ends when all cells are filled. The payoff depends only on the final position, independently of the order in which the cells were occupied. The turn order is random: in each round a fair coin flip decides which player makes the next move. In contrast to the selection games from [12], players are not restricted to the placement of pieces of the own color. Each player is allowed to play either "0" or "1".

We prove that for each real-valued payoff function such a board filling game with random-turn order is MC-perfect. An interesting corollary is that for each position the best moves for players A and B are coupled: When an optimal move for A is to place a "0" in cell i, then it is an optimal move for B to place a "1"

in the very same cell i. And, vice versa, when an optimal move for A is to place a "1" in some cell j, then it is an optimal move for B to place a "0" in just this cell j.

The paper is organized as follows. In Section 2 the basic theorem and its proof are given. Section 3 contains example games and generalisations of the basic result (like veto moves and history-dependence). Readers who are mainly interested in applications and not so much in abstract theory, may jump directly to the concrete parts of Section 3, namely to the descriptions of the games OdOku (in Subsection 3.1) and TOP FOUR (in 3.2). Section 4 concludes and gives four open problems.

2 The Basic Result for Board-Filling Games

Two players A and B play a zero-sum game with finitely many moves. The board is given by n cells. At the end of the game all cells are filled by values from $\{0,1\}$, and the payoff is given by a real-valued function $f : \{0,1\}^n \rightarrow \mathbb{R}$.

$f(X)$ is what B has to pay to A when the game ends in full board position X. So, A wants to maximize $f(X)$ in the end, or the expected value of it for partially filled board positions X. Analogously, B wants to minimize $f(X)$ or the expected value of it for partially filled positions.

In the beginning all n cells $x(1), ..., x(n)$ are empty. The game consists of n moves. Each move has the following structure with two parts. First, a fair coin is flipped. The outcome decides which player is to act in the second part of this move. Second, the winner of the coin flip fills either a "0" or a "1" into an empty cell. It is his choice, which cell and which value.

It turns out that optimal play is coupled to the outcome of randomly filling the cells. To prepare for this result, we define

$Ef[X] =$ expected f-value for position X, when the remaining free cells in X are filled up randomly and independently of each other with 0 and 1 (probability $\frac{1}{2}$ for each value at each cell).

Observe that the real-valued function $Ef[X]$ is well-defined not only for full-board positions X but for all X in the set $\{-,0,1\}^n$ of intermediate game positions, where "$-$" stands for an empty cell in the board. The expected f-values exist as n is finite and f is defined on the whole set $\{0,1\}^n$ of potential final positions. Analogously, $Ef[X0(i)]$ and $Ef[X1(i)]$ are the conditional expectations for the positions $X0(i)$ and $X1(i)$, where cell i has obtained value 0 and 1, respectively. When we randomly fill up the whole board, we can start with any free cell i, thus arriving at

$$Ef[X] = 0.5 \cdot Ef[X0(i)] + 0.5 \cdot Ef[X1(i)]. \tag{1}$$

This is equivalent to

$$0.5 \cdot Ef[X0(i)] = Ef[X] - 0.5 \cdot Ef[X1(i)] \tag{2}$$

and

$$0.5 \cdot Ef[X1(i)] = Ef[X] - 0.5 \cdot Ef[X0(i)]. \tag{3}$$

Assuming the game described above and optimal play by A and B in all positions, we define another function of expected f-values, namely

$E\text{Opt}_f[X]$ = expected f-payoff to A, when the game starts in position X, and A and B act optimally in all possible positions.

With a similar argument as for $Ef[X]$ one sees that $E\text{Opt}_f[X]$ is well-defined on the set $\{-, 0, 1\}^n$ of intermediate game positions, taking in account additionally the minimax property of 2-player zero-sum games. Observe that $E\text{Opt}_f[X]$ is the expected payoff BEFORE the coin flip decides which player moves in X. Analogously, $E\text{Opt}_f[X0(i)]$ and $E\text{Opt}_f[X1(i)]$ are defined for all cells i that are still empty in position X.

Theorem 1

$$E\text{Opt}_f[X] = Ef[X] \tag{4}$$

for all intermediate game positions X *in* $\{-, 0, 1\}^n$.

Proof. The proof works by induction in the number k of free cells. For $k = 0$, nothing has to be shown as X is a final position and thus $Ef[X] = f(X)$, and $E\text{Opt}_f[X] = f(X)$.

Now assume $k > 0$, and the statement of the theorem has been proved for all positions with at most $k-1$ free cells. When there are k free cells left (call them cell 1 to cell k, without loss of generality), the player winning the coin flip has $2 \cdot k$ choices for his move: putting 0 to cell 1, putting 1 to cell 1, putting 0 to cell 2, ..., putting 1 to cell k. As each of the players has probability $\frac{1}{2}$ to execute the next move in position X, we obtain

$$\begin{aligned} E\text{Opt}_f[X] = \quad & 0.5 \cdot \max\left\{E\text{Opt}_f[X0(i)], E\text{Opt}_f[X1(i)]; i = 1, 2, ...k\right\} \\ & + 0.5 \cdot \min\left\{E\text{Opt}_f[X0(i)], E\text{Opt}_f[X1(i)]; i = 1, 2, ...k\right\}. \end{aligned}$$

The max-expression in the upper line comes from the case where player A is to move. The min-expression in the lower line comes from the options for player B. By induction hypothesis we have $E\text{Opt}_f[X0(i)] = Ef[X0(i)]$ and $E\text{Opt}_f[X1(i)] = Ef[X1(i)]$ for $i = 1, ...k$. Hence

$$\begin{aligned} E\text{Opt}_f[X] = \quad & \max\{0.5 \cdot Ef[X0(i)], 0.5 \cdot Ef[X1(i)]; i = 1, 2, ...k\} \\ & + \min\{0.5 \cdot Ef[X0(i)], 0.5 \cdot Ef[X1(i)]; i = 1, 2, ...k\}. \end{aligned} \tag{5}$$

Substituting according to (2) and (3) in the min-part of this expression, we have

$$\begin{aligned} \min\{...\} &= \min\{Ef[X] - 0.5 \cdot Ef[X1(i)], Ef[X] - 0.5 \cdot Ef[X0(i)]; i = 1, 2, ...k\} \\ &= Ef[X] + \min\{-0.5 \cdot Ef[X1(i)], -0.5 \cdot Ef[X0(i)]; i = 1, 2, ...k\} \\ &= Ef[X] - \max\{0.5 \cdot Ef[X1(i)], 0.5 \cdot Ef[X0(i)]; i = 1, 2, ...k\}. \end{aligned}$$

The crucial point here is that always $\min\{-a, -b, ...\} = -\max\{a, b, ...\}$.
Resubstituting into (5) gives

$$\begin{aligned} E\mathrm{Opt}_f[X] = & \quad 0.5 \cdot \max\{Ef[X0(i)], Ef[X1(i)]; i = 1, 2, ...k\} + Ef[X] \\ & - 0.5 \cdot \max\{Ef[X0(i)], Ef[X1(i)]; i = 1, 2, ...k\} \\ = & Ef[X]. \end{aligned}$$

This completes the proof of Theorem 1.

Corollary 2. *Let be given any intermediate position X in $\{-, 0, 1\}^n$. When it is optimal for player A to fill cell i, then it is also optimal for player B to fill cell i, but with the complementary value.*

Proof. In Equation (5), let be $\max\{...\} = 0.5 \cdot Ef[X0(j)]$ for some j. Then $\min\{...\} = 0.5 \cdot Ef[X1(j)]$ for the same j. And, vice versa, if $\max\{...\} = 0.5 \cdot Ef[X1(j)]$ for some j, then $\min\{...\} = 0.5 \cdot Ef[X0(j)]$ for the same j. Or somewhat more elaborate in words: When some position X is given and filling cell i with "0" increases the expected payoff for player A by margin $\varepsilon(i)$, then filling cell i with "1" decreases the expected payoff for player A by the very same margin $\varepsilon(i)$. The best move for A is to fill some cell with maximum positive change in the expected payoff - hence the best move for B is to fill the very same cell with the complementary value. This completes the proof of Corollary 2.

Exactly this argument can also be used to generalize Theorem 1. It is not a necessary condition that all free cells are allowed to be filled in the next move. Only a symmetry between the options of the players has to be guaranteed: When player A is allowed to fill some cell i with "1" then player B must be allowed to fill this cell with "0", and vice versa. Of course, in each non-final situation during the game there has to be at least one admissible cell to fill. In Subsection 3.6 some possible directions of generalisation are listed.

Corollary 3 (MC perfectness). *Every board filling game with random turn order is MC-perfect.*

Proof. Given any intermediate position X, the expected outcomes $Ef[X0(i)]$ and $Ef[X1(i)]$ for random fillings are well defined, as each full board has a finite f-value by definition of f. Hence, traditional Monte Carlo sampling (see for instance [9] or [10]) for each such position converges to the expected value. So, for simulation parameter T going to infinity, only cells with optimal expected value will be proposed by pure Monte Carlo. (Observe that the cell with optimal value need not be unique.) This completes the proof of Corollary 3.

3 Examples and Generalisations

Below we provide four examples (3.1 to 3.4) and give two generalisations (3.5 and 3.6).

3.1 The Random-Turn Game OdOku

OdOku is a new MC-perfect game for two players [3].

Rules: Players A and B move in random order. The second part of its name ("dOku") indicates that OdOku is related to Sudoku. The board is quadratic with 5×5 cells. As seen in the left part of Figure 2, it is divided into five groups, one central cross, and four outer groups. There are five plus five more groups: one for each row, and one for each column. So, altogether 15 groups.

In each turn the player to move puts a 0 or a 1 in any free cell. When all cells of the board are filled, for each group its balance or unbalance is determined. A group is balanced when it contains two or three "1" (and three or two "0", respectively), and unbalanced otherwise. Player B ("B" like in **b**alanced) wants to maximize the number of balanced groups. Player A ("A" like **a**nti-balanced) wants to minimize the number of balanced groups. The right part of Figure 2 shows the end position of an OdOku match. The score is 8. Random OdOku games result in an average score of 9.375 balanced groups.

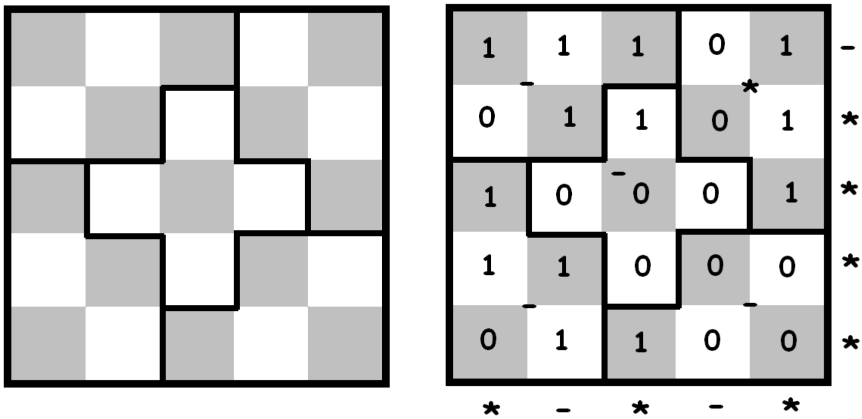


Fig. 2. The diagram on the left side shows an empty OdOku board. The diagram on the right shows the final position of an OdOku game. For each of the 15 groups "*" or "-" sign indicate if this group is balanced or not. Eight of the 15 groups are balanced.

Like all games with random turn order, OdOku can suffer from match instances, where the coin severely favors one of the players. One possible counter measure is to play OdOku simultaneously on two boards I and II. The first player has role A on board I, and role B on board II. The other player complements this by being B on board I and A on board II. When the coin shows "head", each player makes a move on the board where he is A. At "tail", the players move on the other board in their roles as B. At the end the scores of the two boards are compared to find the overall winner. Synchronous execution of the moves guarantees that a player cannot use a mirror strategy.

O-36	X-39	X-37	O-33	X-13	X-42	O-41
X-35	X-31	X-15	X-19	O-12	O-26	O-40
X-34	O-30	O-14	X-18	O-11	O-25	O-38
X-28	X-29	X-10	O-17	O-9	X-24	O-32
X-27	O-20	X-8	X-6	O-7	X-22	O-23
O-5	O-4	O-2	O-1	X-3	O-16	O-21

Fig. 3. A final position of a TOP FOUR game. The number in each cell indicates in which move this cell was filled. Player X wins, because his highest quad consists of $(X-35, X-31, X-15, X-19)$, with height 5. Opponent O has three equally highest quads, namely $(O-33, O-12, O-25, O-32)$, $(O-41, O-26, O-11, O-17)$, and $(O-41, O-40, O-38, O-32)$, with height 4.5.

3.2 The Game TOP FOUR

TOP FOUR was proposed by this author in July 2011. It is a random-turn variant of the popular game "Connect Four" [13]. Given the standard vertical grid of size 7×6, the two players drop pieces in the grid (like in Connect4), until the grid is completely filled (unlike in Connect4). The object of the game is to obtain four pieces of the own color connected: either vertically, or horizontally, or diagonally. Such a connected group is called "good quad" or "quad" for short. When all 42 positions are occupied, the outcome is determined: If there is no quad at all, the game is a draw. Otherwise the side with the "highest" quad is declared winner. The height of a quad is the height of its center of gravity. Figure 3 shows a final position of a game between players O and X. If the highest quads of both players have the same height, a tiebreak rule is used: A quad with center of gravity more to the left of the grid is better. When there is still a tie, then the quad for which the highest cell is more to the left, makes the winner.

Games with random turn order carry the risk that the game is decided early, because one side had more opportunities to move than the opponent. In TOP FOUR this danger is not so expressed because an early quad typically has a deep center of gravity, giving room for "high" conters. L. Bremer made a complete Branch-and-Bound analysis to determine the probability that a random filling

of the grid does not result in any quad. This probability for a draw is very small, namely slightly less than 0.072 percent.

TOP FOUR does not fulfill all the conditions of Section 2: in each column of the board only the lowest free cell may be played. However, Theorem 1 and its proof together with the remark between Corollary 2 and Corollary 3 can be applied, as in each position both players have the same set of cells for playing. So, TOP FOUR is MC-perfect.

3.3 SAT Games with Random Turn Order

A satisfiability game (called "SAT game" for short) with random turn order is defined as follows. Given is a Boolean formula over n variables with (several) disjunctive clauses. The players assign truth values to the variables. Player A wants to maximize the number of clauses that are true in the end. Player B wants to minimize this number. A move consists in fixing a "free" variable to one of its possible values "true" or "false". The turn order is random, so in each step the player to move wants to maximize or minimize the expected number of true clauses, respectively. The game has n moves, one for each free variable. In a weighted version, for each clause j a specific weight $w(j)$ from the real numbers is given. The players want to maximize and minimize, respectively, the sum of the weights of those clauses that are true. Several normal board filling games can be formulated as SAT games. Allowing not only disjunctive clauses but Boolean formulae in general, any deterministic game with outcomes in $\{0, 1\}$ can be encoded by an appropriate Boolean formula. Our Theorem 1 states that SAT games with random turn order are MC-perfect.

3.4 Strange Games Related to Mathematical Problems

Each cell of the game board may stand for a coefficient of a mathematical problem.

- As an example with a one-dimensional set of variables, assume a polynomial of degree n with leading coefficient 1: $P(x) = x^n + a_{(n-1)} \cdot x^{(n-1)} + ... + a_1 \cdot x + a_0$. The players fix the a_i, for instance to values in $\{+1, -1\}$. At the end a certain property of the polynomial $P(x)$ decides, who is winner. This might be, for instance, the question if $P()$ has at least k zeros on the real axis or not.
- As an example with a two-dimensional array think of an $m \times m$ matrix M, so there are $n = m^2$ cells. Player A wants to maximize the absolute size of the third-largest eigen-value of M, whereas player B wants to keep it small.

Pure Monte Carlo with a large simulation parameter T will play such games almost perfectly. However, theoretical insights from mathematics can give perfect play with only a finite (and often small) amount of computations. An example is OdOku from Subsection 3.1 above. Having in mind the additivity of expectations and the fact that in OdOku the payoff function f is the sum of 15 elementary

functions (one for each group), one can locally, for each free cell, determine the expected contribution of this cell to the final score. A cell that is locally best is also globally best and should be played.

3.5 (p,q,r,s)-Generalisation

Most of the results on selection games in [12] do not only hold for random turn orders with FAIR coin flips, but also for any parameter p, $0 < p < 1$, such that player A obtains the right to move with probability p, and opponent B the right with $1 - p$. However, our result for board filling games does not hold in such generality. The best we were able to achieve is the following setting which looks somewhat technical: It relies on two distinguishable fair or non-fair coins for the determination of the next player and his options.

1. Throwing "00" means: A has to place a 1 somewhere.
2. Throwing "01" means: A has to place a 0 somewhere.
3. Throwing "10" means: B has to place a 1 somewhere.
4. Throwing "11" means: B has to place a 0 somewhere.

Let the probabilities for the cases $00, 01, 10, 11$ be given by probabilities $p \cdot r$, $q \cdot s$; $q \cdot r$, $p \cdot s$, respectively. Here p, q, r, s, are real numbers between 0 and 1, such that $p + q = 1$ and $r + s = 1$. The analogues of the theoretical results from Section 2 hold. The idea of the proof is to group the moves in two classes: ("A filling 0" relates to "B filling 1") and ("A filling 1" relates to "B filling 0"). In each class the max/min arguments from Section 2 can be applied.

We give two examples for possible choices of the parameters. The first one is rather artificial and has $p = 2/5, q = 3/5; r = 1/3, s = 2/3$. This leads to move probabilities

- 2/15 for A to place a 1.
- 6/15 for A to place a 0.
- 3/15 for B to place a 1.
- 4/15 for B to place a 0.

The second example is an important special case: $q = 0$, thus $p = 1$; and $r = s = 0.5$. This means that A gets the right to move with 0.5, and has to place a 1. Analogously, B also gets the right to move with 0.5, and has to place a 0. These games are just the "selection games" of [12].

3.6 Generalisations of the Game Class

Partially Ordered Cells. More general than in TOP FOUR, the set of cells can be partially ordered in arbitrary ways. The meaning is that a cell can be filled only when all cells below it are filled already.

Arbitrary Dependance on Game History. It may depend on the history of a game so far which cells are allowed to be filled. So, not only the current position on the board, but also the order in which and how the cells were filled may determine which cells are allowed in the next move. Examples are condensation games (the players build crystals, starting from some original seed cells), games with gravity (like Connect 4), games with spread conditions (the next move has to be "far" away from the most recent moves). Observe the different roles of history for feasibility and evaluation: the feasibility of a cell in a certain position may depend arbitrarily on the history of the game, but the evaluation at the end of the game does not.

Veto Cells, Marked by the Opponent. In this class, a move consists of four parts.

1. By some procedure (maybe involving actions of the two players, chance, influence of referees or spectators) the current veto parameter k is determined. k has to be an integer strictly smaller than the number of free cells on the board.
2. A fair coin flip decides for the next move which player makes the vetoes and which player (the other one) has to fill a cell.
3. The veto player forbids k of the free cells.
4. The other player occupies one of the non-forbidden free cells.

In analogy to Theorem 1 the theoretical result is: The games are MC-perfect; optimal veto cells and optimal fills are in symmetry again.

Detailed proofs for these generalisations as well as example games will be presented in a forthcoming paper.

4 Conclusions and Open Questions

Board filling games with random turn order are Monte Carlo perfect. In each position, the optimal moves for the two players are directly related: When the optimal choice for player A is to fill a "1" into cell i, then the optimal move for player B is to fill a "0" into cell i, and vice versa.

We are interested in three open questions, concerning algorithmic aspects of Monte Carlo perfect games.

1. Do there exist games with random turn order where UCT [11] or some other refined MCTS algorithm [8] does not perform as well as pure Monte Carlo, when given the same FINITE amount of computing time?
2. The AMAF heuristic (AMAF standing for "All Moves As First") is used in many MCTS algorithms. How much does AMAF improve the performance of pure Monte Carlo when used for MC-perfect board filling games?
3. Does pure Monte-Carlo in random-turn games also exhibit the phenomena of laziness (playing "less concentrated" when clearly ahead or clearly behind) and self-play basins (performance difference between $MC(T)$ and $MC(2 \cdot$

T) being largest for some intermediate parameter T)? So far, experimental evidence has been collected only for games with alternating turns, see [1] and [2].

Moreover, an open philosophical question may arise. From the viewpoint of game design it reads as follows. Is a board filling game with random turn order interesting for humans, when human players "typically" have difficulties to find best cells in "typical" positions? Independently of the answer we make a call for creativity: Find interesting board filling games with random turn order! In our eyes, OdOku and TOPFOUR are not bad, but there is still room for improvement. The applet "Hexamania" from D. Wilson's site [14] allows to get an impression how almost-perfect matches in random-turn Hex and random-turn Tripod look like. In the human scene for playing board games, games with random turn order so far live only in a very small niche. The pirate game "Blackbeard" [4] may be the most prominent example - with rather mixed evaluations by the users on BoardGameGeek.

Acknowledgments. Most connection games can be interpreted as board filling games. Thanks go to Cameron Browne for several lively discussions and for his groundbreaking work on connection games [5]. Thanks also to Yi Yang, Shuigeng Zhou, and Yuan Li. Their presentation of the connection game Square++ [15] was the first time I heard about theoretical analysis of a game with random turn order. Yi Yang also pointed me to the seminal paper on random-turn games by Peres, Schramm, Sheffield, and Wilson. Thanks to David Wilson for his nice computer program Hexamania that plays random-turn Hex and Tripod. Thanks to Lars Bremer for computing the probability that the game TOP FOUR ends in a draw, and thanks to Lisa Schreiber for test playing and helpful comments. Finally, three anonymous referees, the editors, and the audience in the conference "Advances in Computer Games 13" had helpful questions and remarks, leading to a clearer presentation. From Bela Bollobas I took the picturesque formulation with the "first glance" praise in the introduction.

References

1. Althöfer, I.: On the Laziness of Monte-Carlo Game Tree Search in Non-Tight Situations. Technical report (2008), http://www.althofer.de/mc-laziness.pdf
2. Althöfer, I.: Game Self-Play with Pure Monte-Carlo: the Basin Structure. Technical report (2010), http://www.althofer.de/monte-carlo-basins-althoefer.pdf
3. Althöfer, I.: Rules and Sample Match of the Game OdOku. Technical report (2011), http://www.althofer.de/odoku.html
4. Berg, R.H.: Pirate board game, Blackbeard. Avalon Hill Company (1991), evaluated, http://www.boardgamegeek.com/boardgame/235/blackbeard (accessed on August 01, 2011)
5. Browne, C.: Connection Games: Variations on a Theme. AK Peters, Massachusetts (2005)
6. Browne, C.: On the Dangers of Random Playouts. ICGA Journal 34, 25–26 (2011)

7. Brügmann, B.: Monte Carlo Go. Unpublished Report (1993), but online available at several places, for instance at http://www.althofer.de/Bruegmann-MonteCarloGo.pdf
8. 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)
9. Fishman, G.S.: Monte Carlo: Concepts, Algorithms, and Applications. Springer, New York (1995)
10. Hammersley, J.M., Handscomb, D.C.: Monte Carlo Methods. Methuen, London (1975)
11. 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)
12. Peres, Y., Schramm, O., Sheffield, S., Wilson, D.B.: Random-Turn Hex and other Selection Games. American Mathematical Monthly 114, 373–387 (2007)
13. Tromp, J.: John's Connect Four Playground, http://homepages.cwi.nl/~tromp/c4/c4.html (last accessed on August 06, 2011)
14. Wilson, D.B.: Hexamania - Computer Program for Playing Random-Turn Hex, Random-Turn Tripod, and the Harmonic Explorer Game (2005), http://dbwilson.com/hex/
15. Yang, Y., Zhou, S.G., Li, Y.: Square++: Making a Connection Game Win-Lose Complementary, Free Deadlock and Playing-Fair. Technical Report, School of Computer Science, Fudan University (2010)

Modeling Games with the Help of Quantified Integer Linear Programs*

Thorsten Ederer, Ulf Lorenz, Thomas Opfer, and Jan Wolf

Institute of Mathematics, Technische Universität Darmstadt, Germany

Abstract. Quantified linear programs (QLPs) are linear programs with mathematical variables being either existentially or universally quantified. The integer variant (Quantified linear integer program, QIP) is PSPACE-complete, and can be interpreted as a two-person zero-sum game. Additionally, it demonstrates remarkable flexibility in polynomial reduction, such that many interesting practical problems can be elegantly modeled as QIPs. Indeed, the PSPACE-completeness guarantees that all PSPACE-complete problems such as games like Othello, Go-Moku, and Amazons, can be described with the help of QIPs, with only moderate overhead. In this paper, we present the *Dynamic Graph Reliability* (DGR) optimization problem and the game *Go-Moku* as examples.

1 Introduction

Game playing with a computer fascinates people all over the world since the beginnings of the Personal Computer, the computer for everyone. Thousands of boys and girls loved to play Atari's Pong or Namco's Pacman and many other games. The Artificial Intelligence community has picked up the scientific aspects of game playing and provided remarkable research results, especially in the area of game tree search. One example of a successful research story is coupled to the game of chess [24,3,14,13,12,5]. Algorithmic achievements like the introduction of the Alphabeta algorithm or the MTD(f) [1] algorithm [20] play a keyrole for the success. Currently, we observe a similar evolution in Computer Go [23]. Here, the UCT-algorithm [2] dominates the search algorithm. Interestingly, it arose from a fruitful interplay of Theory and Practice [16,4]. In Go, the machines increase their playing strength year by year. Some other interesting games have been completely solved [11], in others the machines dominate the human players [10].

Independently, in the 1940s, linear programming arose as a mathematical planning model and rapidly found its daily use in many industries. However, integer programming, which was introduced in 1951, became dominant far later at the beginning of the 1990s.

For traditional deterministic optimization one assumes data for a given problem to be fixed and exactly known when the decisions have to be taken. Nowadays, it is possible to solve very large mixed integer programs of practical size, but companies observe

* Research partially supported by German Research Foundation (DFG) funded SFB 805.

[1] MTD(f) or MTD(f,n): Memory-enhanced Test Driver with node n and value f.

[2] UCT: Upper Confidence bounds applied to Trees.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 270–281, 2012.
© Springer-Verlag Berlin Heidelberg 2012

an increasing danger of disruptions, i.e., events occur which prevent companies from acting as planned. Data are often afflicted with some kinds of uncertainties, and only estimations, maybe in form of probability distributions, are known. Examples are flight or travel times. Throughput-time, arrival times of externally produced goods, and scrap rate are subject to variations in production planning processes.

Therefore, there is a need for planning and deciding under uncertainty which, however, often pushes the complexity of traditional optimization problems, that are in P or NP, to PSPACE. These new problems contain the spirit of games more than the spirit of classic optimization. An interesting result from Complexity Theory is that all NP-complete problems can be converted into each other with only moderate overhead. Analogously, all PSPACE-complete problems can be converted into each other with little effort, and the quantified versions of integer linear programs [25] cover the complexity class PSPACE. As a consequence, we can model every other problem which is PSPACE-complete or easier, with moderate (i.e., polynomial) overhead, and thus, a PSPACE complete game such as Othello, Go-Moku, and Amazons[3] is modeled via the game's rules which are encoded. It is not necessary to encode the game tree of such a game. Chess, Checkers, and Go do not necessarily belong to this group of games. Their inspected extensions are even EXPTIME-hard. For the interested reader, we refer to [7,22]. If we restrict the maximum number of allowed moves in these games by a polynomial in the input size, they are in PSPACE, and at least checkers is then PSPACE complete [8].

Suitable candidate algorithms for solving QIPs are the typical algorithms from AI such as Alphabeta, UCT, Proof Number Search (PNS) [26] or MTD(f). Because of the transformation overhead from the original problem description to a QIP, there is a certain loss in solution performance. We believe that this overhead might be accepable, because solving QLPs allows the fast generation of bounds to the original QIP. We are optimistic that these QLP bounds can considerably speed up the search process for the QIP, similarly as the solutions of LP-relaxations speed up the branch and bound algorithm in the context of conventional linear integer programming. However, to answer this question is a matter of ongoing research. In summery, three subtasks must be solved in order to generate huge impact on both, Game Playing and Mathematical Optimization under Uncertainty.

- Applicability must be shown. This means we have to convince that relevant problems can elegantly be modeled with the help of QIPs.
- Fast algorithms for QLPs, the relaxed versions of QIPs, must be found.
- It has to be shown that the QLP solutions can be used in order to speed up the search processes of QIPs by a huge margin. Note that this last step took about 20 years for conventional Mathematical Programming.

Thus, the idea of our research is to explore the abilities of linear programming when applied to PSPACE-complete problems, similar as it was applied to NP-complete problems in the 1990s. We could already show that QLPs have remarkable polyhedral properties [18]. Moreover, we are able to solve comparably large QLPs in reasonable time

[3] For an overview, cf. http://en.wikipedia.org/wiki/Game_complexity [21,9,15].

[6]. Recent research has shown that our QLP algorithm can be improved further by adopting some techniques known from the gaming community. For example, we could derive cutting planes with an interesting similarity to alpha-beta pruning, which reduce the search time by half.

2 The Problem Statement: Quantified Linear Programs

2.1 Problem Statement

In the following, the definition of a *Quantified Linear Program* is stated. Let $\mathbb{Q}$ describe the set of rational numbers and $\mathbb{Z}$ the set of integer numbers.

Given: A vector of variables $x = (x_1, ..., x_n) \in \mathbb{Q}^n$, upper and lower bounds $u \in \mathbb{Z}^n$ and $l \in \mathbb{Z}^n$ with $l_i \leq x_i \leq u_i$, a matrix $A \in \mathbb{Q}^{m\times n}$, a vector $b \in \mathbb{Q}^m$ and a quantifier string $Q = (q_1, ..., q_n) \in \{\forall, \exists\}^n$, where the quantifier q_i belongs to the variable x_i, for all $1 \leq i \leq n$.

We denote a QIP/QLP as $[Q(x) : Ax \leq b]$. A maximal subset of $Q(x)$, which contains a consecutive sequence of quantifiers of the same type, is called a (variable) *block*. A full assignment of numbers to the variables is interpreted as a game between an existential player (fixing the existentially quantified variables) and a universal player (fixing the universally quantified variables). The variables are set in consecutive order, as determined by the quantifier string. Consequently, we say that a player makes the move $x_k = z$, if he[4] fixes the variable x_k to the value z. At each such move, the corresponding player knows the settings of $x_1, ..., x_{i-1}$ before setting x_i. If, at the end, all constraints of $Ax \leq b$ hold, the existential player wins the game. Otherwise the universal player wins.

Problem Statement: Is there an assignment for variable x_i with the knowledge, how $x_1, ..., x_{i-1}$ have been set before, such that the existential player wins the game, no matter, how the universal player acts when he has to move?

The problem occurs in two variants: (a) all variables are discrete (QIP) and (b) all variables are rational (QLP).

2.2 Solutions of QIPs and QLPs: Strategies and Policies

Definition 1. (Strategy) *A* strategy *for the existential player S is a tuple $(V_x\dot{\cup}V_y, E, L)$, that is, a labeled tree of depth n with vertices V and edges E ($|V|$ and $|E|$ being their respective sizes), where V_x and V_y are two disjoint sets of nodes, and $L \in \mathbb{Q}^{|E|}$ is a set of edge labels. Nodes from V_x are called* existential nodes*, nodes from V_y are called* universal nodes. *Each tree level i consists either of only existential nodes or of only universal nodes, depending on the quantifier q_i of the variable x_i. Each edge of the set E, leaving a tree-node of level i, represents an assignment for the variable x_i. $l_i \in L$ describes the value of variable x_i on edge $e_i \in E$. Existential nodes have exactly one successor, universal nodes have as many successors as the universal player has choices at that node.*

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

In case of QLPs, it suffices to deal with two successors, induced by the upper and the lower integer bounds of a universal variable, below each universal node [18]. A strategy is called a winning strategy, *if all paths from the root to a leaf represent a vector x such that $Ax \leq b$.*

Definition 2. (Policy) *A* policy *is an algorithm that fixes a variable x_i, being the ith component of the vector x, with the knowledge, how $x_1, ..., x_{i-1}$ have been set before.*

A three-dimensional example of a QIP/QLP is given below:

$$\forall x_1 \in [-1, 0]\, \forall x_2 \in [0, 1]\, \exists x_3 \in [-2, 2] : \begin{pmatrix} 10 & -4 & 2 \\ 10 & 4 & -2 \\ -10 & 4 & 1 \\ -10 & -4 & -1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} 0 \\ 4 \\ 12 \\ 8 \end{pmatrix}$$

If we restrict the variables in the example to the integer bounds of their domains, we observe a winning strategy for the existential player as shown in Figure 1. In this example, a + in a tree leaf means that the existential player wins when this leaf is reached. A - marks a win for the universal player. Numbers at the edges mark the choices for variables. If the universal player moves to -1 and 0 (i.e., he sets $x_1 = -1$ and $x_2 = 0$) the existential player has to move to 2. If the universal player moves to -1 and 1, the exis-

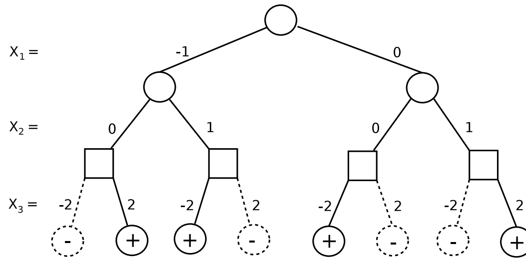


Fig. 1. The winning strategy (solid) for the integer QLP example above

tential player must set the variable $x_3 = -2$ etc. We see that the existential player has to react carefully to former moves of the universal player. If we now relax the variables, allowing non-integral values for the corresponding domains, the resulting solution space of the corresponding QLP becomes polyhedral. Moreover, the solution of the resulting problem when $x_1 = -1$ is a line segment: B (cf. Fig. 2). On the left side of Figure 2, we can still see the corresponding partial strategy of the integer example in Figure 1. The strategy consists of the two end-points of B. On the right side of Figure 2, the same convex hull of the solution space is shown from another perspective. We observe that the existential player has more freedom in the choice of x_3, when the universal player sets $x_1 = 0$. If $x_1 = 0$, the solution space of the rest problem will just be the facet C.

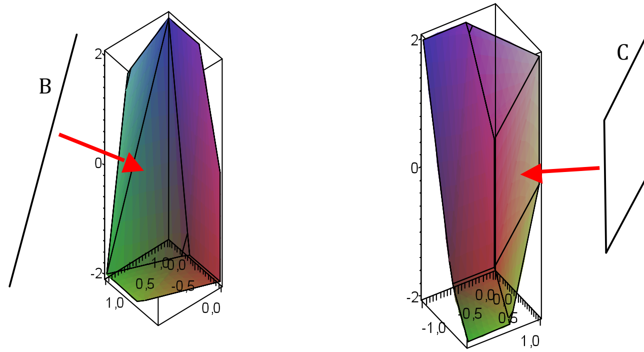


Fig. 2. A visualization of the 3-d solution space of the example above

Remark: If the integrality constraints of a QIP are relaxed, this will result in a QLP. One may distinguish between QLPs where only the variables of the existential player are relaxed, and QLPs where all variables are relaxed. However, these two variants are equivalent to each other, as in [18] it was shown that the following holds.

– Let $y_1, .., y_m$ be the universal variables of a given QLP. Let $l_1, ..., l_m$ be lower bound restrictions to the y-variables and let $u_1, ..., u_m$ be the corresponding upper bounds. The existential player has a winning strategy against the universal player who takes his choices $y_i \in \{l_i, u_i\}, i \in 1...m$ if and only if the existential player has a winning policy against the universal player who takes his choices from the corresponding rational intervals, i.e., $y_i \in [l_i, u_i], i \in 1...m$.
– Whether or not there is a winning strategy for the existential player, it can be determined with polynomially many bits, as long as the number of quantifier changes is constant, independently of the number of variables.

3 Modeling with QIPs

In this section two different games are modeled with the help of QIPs. The first one is a straightforward graph game, where a person has to travel on a graph from a given source node to a desired target node. While he is traveling, an (evil) opponent erases some edges and thus destroys the graph. However, also the opponent must follow certain rules. The second game, which is inspected, is the well known *Five in a Row* game. Both are PSPACE-complete, and can therefore not be modeled on small space by propositional logic or by mixed integer linear programming (except PSPACE=NP).

3.1 A Two-Person Zero-Sum Graph Game

In order to demonstrate the modeling power of QIPs, we first present an example for the so called worst-case Dynamic Graph Reliability problem. It is closely related to the QSAT-problem and to the Dynamic Graph Reliability problem (DGR) [19]. A variant

of this game was re-invented by van Benthem [2] and analyzed by Löding and Rhode [17].

A worst-case DGR (wcDGR) is defined as follows.

Given: A directed acyclic graph $G = (V, E)$ with two special nodes s and t. Moreover, we have defined a mapping $f : (V \times E) \to \{0, 1\}$ with $f(v, e) = 1$ if and only if an opponent is allowed to erase edge e when we arrive at node v. Moreover, the opponent has to follow some rules as well. For some edges, the opponent is not allowed to erase more than one of two specific edges. In other words, there is another mapping $g : E \times E \to \{0, 1\}$ with $g(e_1, e_2) = 1$ if and only if it is allowed to erase both e_1 and e_2.

Question: Is there a strategy, which allows the existential player to reach the target node t from start node s, no matter, how the opponent acts?

The starting point of the wcDGR is a directed acyclic graph (DAG). An example is shown on the left side of Figure 3. There, we assume that an opponent may erase at most one of the edges e_4 and e_5. He can make them fail when we arrive at node v_2 or at node v_3. Anyway, never both edges are allowed to fail, and no other edges can fail. The optimization problem is firstly to make a choice whether to travel via edge e_1 or e_2. Then, the opponent erases none or one of the edges e_4 and e_5. Thereafter, we choose a remaining path to target t, if one exists. If we move from node v_2 to node v_3, our opponent is again allowed to make one of the two edges e_4 or e_5 fail.

Let us introduce variables $x_1, ..., x_5$ for edge-choices. $x_i = 1$ means e_i is chosen for traveling. The first block of constraints encodes the flow constraints of the classic shortest-path problem on graphs. The constraints are applied to the x-variables (Fig. 3,

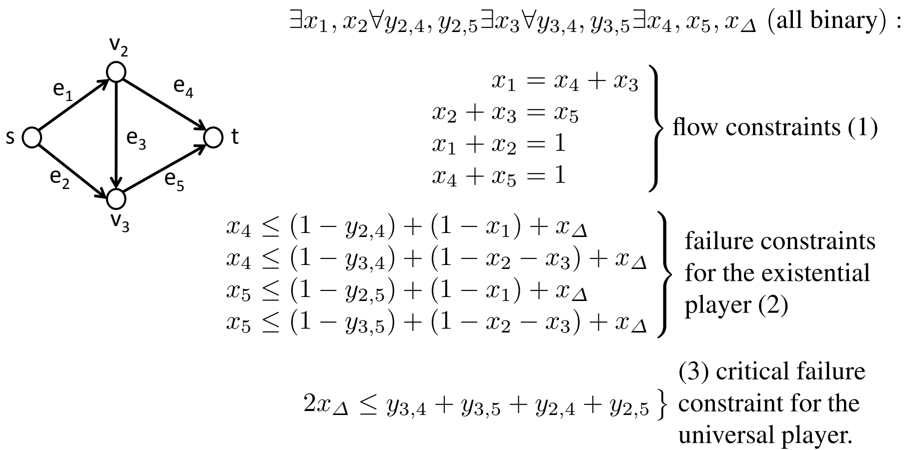
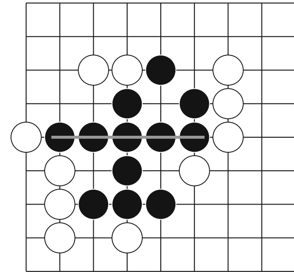


Fig. 3. Graph of wcDGR example and QIP description

right, (1)). $y_{2,4}, y_{3,4}, y_{2,5}, y_{3,5}$ determine whether the opponent makes the edges e_4 or e_5 fail when we reach the nodes v_2 or v_3, i.e., $y_{i,j} = 1$ means that there is a failure on edge e_j. The second block (2) couples the decision variables x_i to the $y_{j,k}$-variables of the opponent. For instance, $x_4 \leq (1-y_{2,4})+(1-x_1)+x_\Delta$ means that the existential player will have to set x_4 to zero and will not be allowed to use edge e_4 if the existential player first moves via edge e_1 and then the universal player sets $y_{2,4} = 1$. Strictly seen, we have to test whether the existential player has moved via node v_2. However, a directed graph can always be pre-manipulated such that all nodes of the original graph can be entered via one specific incoming edge. Of course, for this purpose, additional nodes and edges must be added, in general. The variable x_Δ is used to ensure that also the universal player follows his rules. The last constraint (3) $2x_\Delta \leq y_{3,4}+y_{3,5}+y_{2,4}+y_{2,5}$ expresses that the universal player is constrained by $y_{3,4} + y_{3,5} + y_{2,4} + y_{2,5} \leq 1$. If he breaks this rule, the existential player can set $x_\Delta = 1$ and the constraints of the second block are trivialized. The existential player can then trivially win the game. Last but not least, we can express the problem as shown on the right side of Fig. 3, with the given quantifier-prefix, because the graph of a wcDGR is a DAG and therefore a partial order (in time) of the nodes can be computed.

3.2 Gomoku

Five in a Row or *Gomoku* is a two-person strategy game played with *Go* pieces on a *Go* board. Both players (Black and White, Black begins) alternately place stones. The first player who obtains a row of five horizontally, vertically, or diagonally connected stones is the winner. A once placed stone cannot be moved or removed from the board. With this standard set of rules, it is known that Black always wins on some board sizes (e.g., 15×15 [1]), but the problem is open for arbitrary $n \times n$ boards.



Parameters
As we showed in the preceeding paragraph, we can model that also the universal player has to follow some rules with auxiliary existential variables. We simplify the description of Gomoku by not considering this detail. Instead, we only present the rules for the universal player. Let T denote the set of all moves (there are up to n^2 moves until the board is full) and N^2 the set of all board coordinates. Let H^2 be the set of coordinates of a reduced board (used to detect connected rows of five stones) and C a counting set. The decision parameters δ_t specify, which player places a stone in move t.

$$T := \{1, \ldots, n^2\}, \quad N := \{1, \ldots, n\},$$

$$H := \{1, \ldots, n-4\}, \quad C := \{0, \ldots, 4\},$$

$$\delta_t^b := t \bmod 2, \quad \delta_t^w := 1 - (t \bmod 2)$$

Variables
Let $\mathbb{B} = \{0, 1\}$ denote the binary set. For each move $t \in T$ and each field with coordinates $(i, j) \in N^2$, let there be two binary variables $x^b_{t,i,j}$ indicating a black stone and $x^w_{t,i,j}$ indicating a white stone. Using the set $\{b, w\}$ as upper index implies that a statement is equally valid for both black and white stones.

For each move $t \in T$ and coordinate $(k, l) \in N \times H$, let $h^{\{b,w\}}_{t,k,l}$ be an indicator variable denoting the existence of a horizontally connected black or respectively white row of five stones beginning on this coordinate. The range of column indices is slightly smaller than N, because each such connected row cannot start near the right board edge. Analogously, let there be vertical indicator variables $v^{\{b,w\}}_{t,k,l}$ for each move $t \in T$ and coordinate $(k, l) \in H \times N$ and diagonal indicator variables $d^{\{b,w\}}_{t,k,l}$ for each move $t \in T$ and coordinate $(k, l) \in H^2$.

Using the connected row indicators, winning criteria can be expressed. Let $s^{\{b,w\}}_t$ be a monotonously increasing indicator function (*step function*), which raises the first time a player has any connected row of five. Further, let $p^{\{b,w\}}_t$ be an indicator function with norm one (*peak function*), which peaks the first time a player has any connected row of five and is false otherwise. The event of winning the game can be expressed by indicator variables $e^{\{b,w\}}_t$, which is true if and only if a player achieves any connected row of five for the first time and his opponent did not achieve any connected row of five before. The retaliation indicator $r^{\{b,w\}}$ indicates, if a player does not win at all.

$$x^{\{b,w\}}_{t,i,j} \in \mathbb{B}^{|T \times N^2|}$$

$$h^{\{b,w\}}_{t,k,l} \in \mathbb{B}^{|T \times N \times H|},\ v^{\{b,w\}}_{t,k,l} \in \mathbb{B}^{|T \times H \times N|},\ d^{\{b,w\}}_{t,k,l}, u^{\{b,w\}}_{t,k,l} \in \mathbb{B}^{|T \times H^2|}$$

$$s^{\{b,w\}}_t \in \mathbb{B}^{|T|},\ p^{\{b,w\}}_t \in \mathbb{B}^{|T|},\ e^{\{b,w\}}_t \in \mathbb{B}^{|T|},\ r^{\{b,w\}} \in \mathbb{B}$$

To express the quantifier string in a compact form, we denote the following abbreviations:

$$A_0 := \left\{\forall\,(i,j) \in N^2 : x^{\{b,w\}}_{0,i,j},\ s^{\{b,w\}}_0\right\}$$

$$\begin{aligned} \forall t \in T: \quad A_t := \Big\{&\forall\,(i,j) \in N^2 : x^{\{b,w\}}_{t,i,j},\ \forall\,(k,l) \in N \times H : h^{\{b,w\}}_{t,k,l}, \\ &\forall\,(k,l) \in H \times N : v^{\{b,w\}}_{t,k,l},\ \forall\,(k,l) \in H^2 : d^{\{b,w\}}_{t,k,l}, \\ &s^{\{b,w\}}_t, p^{\{b,w\}}_t, e^{\{b,w\}}_t\Big\} \end{aligned}$$

Optimization Model
The solution of a quantified optimization model is the information, if the player (in this case Black) can win against his opponent. If he can win, the first move leading to the victory is provided. (If one wishes to play a full *Go-Moku* game using a quantified model, one therefore has to solve the model after each move of the opponent.)

We choose to extend the basic model with an objective function to rate Black's strategy. If Black can win at all, the optimal objective value is the first possible move in which Black can win. If Black cannot win, but he can force a draw, the optimal objective value is $(n^2 + 1)$. If Black definitely loses, the optimal objective value minus

$(n^2 + 1)$ depicts the longest possible delay Black can achieve until he is beaten. To extract the basic information if Black can win, the objective value can be compared to the remis value $(n^2 + 1)$.

minimize

$$\sum_{t \in T} t \cdot e_t^b + (2n^2 + 1) \cdot r^b - \sum_{t \in T} t \cdot e_t^w - n^2 \cdot r^w$$

such that

$$\exists A_0 \ \exists A_1 \ \forall A_2 \ \exists A_3 \ \forall A_4 \ \ldots \ \exists \forall \exists \forall \ \ldots \ A_{n^2} \ \exists r^{\{b,w\}}$$

$$\forall (i,j) \in N^2 : \ x_{0,i,j}^{\{b,w\}} = 0, \quad s_0^{\{b,w\}} = 0 \tag{1}$$

(initial state: All intersections are free, both players have not won yet.)

$$\forall t \in T, \ \forall (i,j) \in N^2 : \quad x_{t,i,j}^b + x_{t,i,j}^w \leq 1 \tag{2}$$

(occupation: On every intersection no more than one stone can be placed.)

$$\forall t \in T, \ \forall (i,j) \in N^2 : \quad x_{t,i,j}^{\{b,w\}} \geq x_{t-1,i,j}^{\{b,w\}} \tag{3}$$

(causality: Every once placed stone remains placed.)

$$\forall t \in T : \qquad \sum_{(i,j) \in N^2} x_{t,i,j}^{\{b,w\}} = \sum_{(i,j) \in N^2} x_{t-1,i,j}^{\{b,w\}} + \delta_t^{\{b,w\}} \tag{4}$$

(alternation: In every odd move Black places exactly one stone. In every even move White places exactly one stone. In both cases the opposite player remains idle.)

$$\forall t \in T, \ \forall (k,l) \in N \times H, \ \forall c \in C : \quad h_{t,k,l}^{\{b,w\}} \leq x_{t,k,l+c}^{\{b,w\}} \tag{5a}$$

$$\forall t \in T, \ \forall (k,l) \in N \times H \qquad : \quad h_{t,k,l}^{\{b,w\}} \geq \sum_{c \in C} x_{t,k,l+c}^{\{b,w\}} - 4 \tag{5b}$$

(horizontal rows: a player achieved a horizontal row beginning on coordinate (k, l), if and only if the specified and the next four intersections to the right contain stones of the same color.)

$$\forall t \in T, \ \forall (k,l) \in H \times N, \ \forall c \in C : \quad v_{t,k,l}^{\{b,w\}} \leq x_{t,k+c,l}^{\{b,w\}} \tag{6a}$$

$$\forall t \in T, \ \forall (k,l) \in H \times N \qquad : \quad v_{t,k,l}^{\{b,w\}} \geq \sum_{c \in C} x_{t,k+c,l}^{\{b,w\}} - 4 \tag{6b}$$

(vertical rows: a player achieved a vertical row beginning on coordinate (k, l), if and only if the specified and the next four intersections to the bottom contain stones of the same color.)

$$\forall t \in T, \ \forall (k,l) \in H^2, \ \forall c \in C : \quad d_{t,k,l}^{\{b,w\}} \leq x_{t,k+c,l+c}^{\{b,w\}} \tag{7a}$$

$$\forall t \in T, \ \forall (k,l) \in H^2 \qquad : \quad d_{t,k,l}^{\{b,w\}} \geq \sum_{c \in C} x_{t,k+c,l+c}^{\{b,w\}} - 4 \tag{7b}$$

(downward diagonal rows: a player achieved a downward diagonal row beginning on coordinate (k, l), if and only if the specified and the next four intersections to the bottom-right contain stones of the same color.)

$$\forall t \in T,\ \forall (k,l) \in H^2,\ \forall c \in C: \quad u_{t,k,l}^{\{b,w\}} \leq x_{t,k+c,l+4-c}^{\{b,w\}} \tag{8a}$$

$$\forall t \in T,\ \forall (k,l) \in H^2 \quad : \quad u_{t,k,l}^{\{b,w\}} \geq \sum_{c \in C} x_{t,k+c,l+4-c}^{\{b,w\}} - 4 \tag{8b}$$

(upward diagonal rows: a player achieved an upward diagonal row beginning on coordinate $(k, l+4)$, if and only if this and the next four intersections to the top-right contain stones of the same color. Heed that the upward diagonal row with index (k, l) does *not* contain a stone on intersection (k, l).)

$$\forall t \in T \quad : \quad s_t^{\{b,w\}} \geq s_{t-1}^{\{b,w\}} \tag{9a}$$

$$\forall t \in T,\ \forall (k,l) \in N \times H: \quad s_t^{\{b,w\}} \geq h_{t,k,l}^{\{b,w\}} \tag{9b}$$

$$\forall t \in T,\ \forall (k,l) \in H \times N: \quad s_t^{\{b,w\}} \geq v_{t,k,l}^{\{b,w\}} \tag{9c}$$

$$\forall t \in T,\ \forall (k,l) \in H^2: \quad s_t^{\{b,w\}} \geq d_{t,k,l}^{\{b,w\}} \tag{9d}$$

$$\forall t \in T,\ \forall (k,l) \in H^2: \quad s_t^{\{b,w\}} \geq u_{t,k,l}^{\{b,w\}} \tag{9e}$$

$$\forall t \in T: \quad s_t^{\{b,w\}} \leq s_{t-1}^{\{b,w\}} + \sum_{(k,l) \in N \times H} h_{t,k,l}^{\{b,w\}} + \sum_{(k,l) \in H \times N} v_{t,k,l}^{\{b,w\}} + \sum_{(k,l) \in H^2} d_{t,k,l}^{\{b,w\}} + \sum_{(k,l) \in H^2} u_{t,k,l}^{\{b,w\}} \tag{9f}$$

(row history: The monotonously increasing indicator function s raises in the unique move t, when the player achieves a row for the first time.)

$$\forall t \in T: \quad p_t^{\{b,w\}} = s_t^{\{b,w\}} - s_{t-1}^{\{b,w\}} \tag{10}$$

(critical move: The indicator function p with norm one peaks in the unique move t, when the player achieves a row for the first time.)

$$\forall t \in T: \quad e_t^b \leq p_t^b \tag{11a}$$

$$\forall t \in T: \quad e_t^b \leq (1 - s_t^w) \tag{11b}$$

$$\forall t \in T: \quad e_t^b \geq p_t^b + (1 - s_t^w) - 1 \tag{11c}$$

$$\forall t \in T: \quad e_t^w \leq p_t^w \tag{11d}$$

$$\forall t \in T: \quad e_t^w \leq (1 - s_t^b) \tag{11e}$$

$$\forall t \in T: \quad e_t^w \geq p_t^w + (1 - s_t^b) - 1 \tag{11f}$$

(victory: A player wins in the unique move t, if and only if he achieves his first row in move t and his opponent has not done so before move t. Heed the switched *b/w* indices.)

$$r^{\{b,w\}} + \sum_{t \in T} e_t^{\{b,w\}} = 1 \tag{12}$$

(retaliation: If a player does not win at all, his retaliation indicator is activated. This information is used to weight ties and defeats in the objective function.)

After all, we see that Gomoku could elegantly be described with the help of a QIP. An obvious drawback is that the number of constraints has grown by a factor of n^2. However, it is neither the case that a short description necessarily leads to faster solutions, nor do we claim that our description is the shortest possible one.

4 Conclusion

Quantified Linear Integer Programs form quite a powerful and elegant modeling tool. They have structural properties of Linear Programs on the one hand, but on the other hand, they describe the complexity class PSPACE and therefore are natural candidates for the modeling of games such as graph games, Go-Moku, Sokoban, and many more. We guess that this modeling technique has the potential to bridge the gap between Mathematical Optimization and Game Playing in the Artificial Intelligence. In this paper, we were able to show how to model some games with the help of QIPs. Moreover, we reported about progress solving QLPs. Future work will show whether QLPs can be used to speed up search times for QIPs in a similar way as MIPs are speeded up with the help of LP-relaxation. If QLPs are able to play a similar role for QIPs as LPs play for IPs, we can hope for a performance jump, at least for exact solving of PSPACE-complete games.

References

1. Allis, L.: Searching for solutions in games and artificial intelligence. Ph.D. thesis (1994)
2. van Benthem, J.: An Essay on Sabotage and Obstruction. In: Hutter, D., Stephan, W. (eds.) Mechanizing Mathematical Reasoning. LNCS (LNAI), vol. 2605, pp. 268–276. Springer, Heidelberg (2005)
3. Condon, J., Thompson, K.: Belle chess hardware. In: Clarke, M.R.B. (ed.) Advances in Computer Chess III, pp. 44–54. Pergamon Press (1982)
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. Donninger, C., Lorenz, U.: The Chess Monster Hydra. In: Becker, J., Platzner, M., Vernalde, S. (eds.) FPL 2004. LNCS, vol. 3203, pp. 927–932. Springer, Heidelberg (2004)
6. Ederer, T., Lorenz, U., Martin, A., Wolf, J.: Quantified Linear Programs: A Computational Study. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 203–214. Springer, Heidelberg (2011)
7. Fraenkel, A., Lichtenstein, D.: Computing a perfect strategy for n×n chess requires time exponential in n. J. Comb. Th. A 31, 199–214 (1981)
8. Fraenkel, A.S., Garey, M.R., Johnson, D.S., Schaefer, T., Yesha, Y.: The complexity of checkers on an n×n board. In: 19th Annual Symposium on Foundations of Computer Science (FOCS 1978), pp. 55–64 (1978)
9. Hearn, R.: Amazons is pspace-complete. Tech. Rep. cs.CC/0502013 (February 2005)
10. van den Herik, H., Nunn, J., Levy, D.: Adams outclassed by hydra. ICGA Journal 28(2), 107–110 (2005)

11. van den Herik, H., Uiterwijk, J., van Rijswijk, J.: Games solved: Now and in the future. Artificial Intelligence 134, 277–312 (2002)
12. Hsu, F.H.: Ibm's deep blue chess grandmaster chips. IEEE Micro 18(2), 70–80 (1999)
13. Hsu, F.H., Anantharaman, T., Campbell, M.: No: Deep thought. In: Computers, Chess, and Cognition, pp. 55–78 (1990)
14. Hyatt, R., Gower, B., H.L., N.: Cray blitz. In: Beal, D.F. (ed.) Advances in Computer Chess IV, pp. 8–18. Pergamon Press (1985)
15. Iwata, S., Kasai, T.: The othello game on an n*n board is pspace-complete. Theoretical Computer Science 123, 329–340 (1994)
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. Löding, C., Rohde, P.: Solving the Sabotage Game Is PSPACE-Hard. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 531–540. Springer, Heidelberg (2003)
18. Lorenz, U., Martin, A., Wolf, J.: Polyhedral and algorithmic properties of quantified linear programs. In: Annual European Symposium on Algorithms, pp. 512–523 (2010)
19. Papadimitriou, C.: Games against nature. J. of Comp. and Sys. Sc., 288–301 (1985)
20. Plaat, A., Schaeffer, J., Pijls, W., De Bruin, A.: Best-first fixed-depth game-tree search in practice. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence, vol. 1, pp. 273–279. Morgan Kaufmann Publishers Inc., San Francisco (1995)
21. Reisch, S.: Gobang ist pspace-vollstandig (gomoku is pspace-complete). Acta Informatica 13, 5966 (1999)
22. Robson, J.M.: The complexity of go. In: Proceedings of IFIP Congress, pp. 413–417 (1983)
23. Silver, D.: Reinforcement Learning and Simulation-Based Search in Computer Go. Ph.D. thesis, University of Alberta (2009)
24. Slate, D., Atkin, L.: Chess 4.5 - the northwestern university chess program. In: Frey, P.W. (ed.) Chess Skill in Man and Machine, pp. 82–118. Springer (1977)
25. Subramani, K.: Analyzing Selected Quantified Integer Programs. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 342–356. Springer, Heidelberg (2004)
26. Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: PDS-PN: A New Proof-Number Search Algorithm. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 61–74. Springer, Heidelberg (2003)

Computing Strong Game-Theoretic Strategies in Jotto*

Sam Ganzfried

Computer Science Department,
Carnegie Mellon University
sganzfri@cs.cmu.edu

Abstract. We develop a new approach that computes approximate equilibrium strategies in Jotto, a popular word game. Jotto is an extremely large two-player game of imperfect information; its game tree has many orders of magnitude more states than games previously studied, including no-limit Texas Hold'em. To address the fact that the game is so large, we propose a novel strategy representation called oracular form, in which we do not explicitly represent a strategy, but rather appeal to an oracle that quickly outputs a sample move from the strategy's distribution. Our overall approach is based on an extension of the fictitious play algorithm to this oracular setting. We demonstrate the superiority of our computed strategies over the strategies computed by a benchmark algorithm, both in terms of head-to-head and worst-case performance.

1 Introduction

Developing strong strategies for agents in large games is an important problem in artificial intelligence. In particular, much work has been devoted in recent years to developing algorithms for computing game-theoretic solution concepts, specifically the Nash equilibrium. In two-player zero-sum games, Nash equilibrium strategies have a strong theoretical justification as they also correspond to minimax strategies; by following an equilibrium strategy, a player can guarantee at least the value of the game in expectation, regardless of the strategy followed by his[1] opponent. Currently, the best algorithms for computing a Nash equilibrium in two-player zero-sum extensive-form games (with perfect recall) are able to solve games with 10^{12} states in their game tree [7].

However, many interesting games actually have significantly more than 10^{12} states. Texas Hold'em poker is a prime example of such a game that has received significant attention in the AI literature in recent years; the game tree of two-player limit Texas Hold'em has about 10^{18} states, while that of two-player no-limit Texas Hold'em has about 10^{71} states. The standard approach is to apply an *abstraction* algorithm, which constructs a smaller game that is similar to the

* This material is based upon work supported by the National Science Foundation under grants IIS-0964579, IIS-0905390, and CCF-1101668.

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

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 282–294, 2012.
© Springer-Verlag Berlin Heidelberg 2012

original game; then the smaller game is solved, and its solution is mapped to a strategy profile in the original game [1]. Many abstraction algorithms work by coarsening the moves of chance, collapsing several information sets of the original game into single information sets of the abstracted game (called *buckets*).

In this paper we study a game with many orders of magnitude more states than even two-player no-limit Texas Hold'em. Jotto, a popular word game, contains approximately 10^{853} states in its game tree. However, Jotto does not seem particularly amenable to abstraction in the same way that poker is; we discuss reasons for this in Section 2. Furthermore, even if we could apply an abstraction algorithm to Jotto, we would need to group 10^{841} game states into a single bucket on average, which would almost certainly lose a significant amount of information from the original game. Thus, the abstraction paradigm that has been successful on poker does not seem promising to games like Jotto; an entirely new approach is needed.

We provide such an approach. To deal with the fact that we cannot even represent a strategy for one of the players, we provide a novel strategy representation which we call *oracular form*. Rather than viewing a strategy as an explicit object that must be represented and stored, we instead represent it implicitly through an oracle; we can think of the oracle as an efficient algorithm. Each time we want to make a play from the strategy, we query the oracle, which quickly outputs a sample play from the strategy's distribution. Thus, instead of representing the entire strategy in advance, we obtain it on an as-needed basis via real-time computation.

Our main algorithm for computing an approximate equilibrium in Jotto is an extension of the fictitious play algorithm [3] to our oracular setting. The algorithm outputs a full strategy for one player, and for the other player outputs data such that if another algorithm is run on it, a sample of the strategy's play is obtained. Thus, we can play this strategy, even though it is never explicitly represented. We use our algorithm to compute approximate equilibrium strategies on 2, 3, 4, and 5-letter variants of Jotto. We demonstrate the superiority of our computed strategies over the strategies computed by a benchmark algorithm, both in terms of head-to-head and worst-case performance.

2 Jotto

Jotto is a popular two-player word game. While there are many different variations of the game, we will describe the rules of one common variant. Each player picks a secret five-letter word, and the object of the game is to guess correctly the other player's word first. Players take turns guessing the other player's secret word and replying with the number of common letters between the guessed word and the secret word (the positions do not matter). For example, if the secret word is GIANT and a player guesses PECAN, the other player will give a reply of 2 (for the A and the N, even though they are in the wrong positions). Players often cross out letters that are eliminated and record other logical deductions on a sheet of paper. An official Jotto sheet is shown in Figure 1.

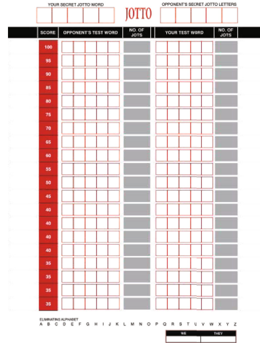


Fig. 1. Official Jotto sheet. Players record guesses and answers, cross off alphabet letters as they become inconsistent, and record other notes and deductions.

Instead of having both players simultaneously guessing the other player's word, we could instead just have one player pick a secret word while the other player guesses it. Let us refer to these players as the *hider* and the *guesser* respectively. If the guesser correctly guesses the word on his k'th try, then the hider gets payoff k while the guesser gets payoff $-k$. This is the variation that we consider in the remainder of the paper.

There are a few limits on the words that the players can select. All words must be chosen from a pre-arranged dictionary. No proper nouns are allowed, and the words must consist of all different letters (some variations do not impose this restriction). Furthermore, we do not allow players to select a word of which other permutations (aka anagrams) are also valid words (e.g., STARE and RATES)[2].

The official dictionary we will be using has 2833 valid 5-letter words. A naïve attempt at determining the size of the game tree is the following. First the hider selects his word, putting the game into one of 2833 states. At each state, the guesser must choose one of 2833 words; the hider gives him an answer from 0-5, and the guesser must now choose one of 2832 words, and so on. The total number of game states will be approximately $2833 \cdot 2833!$.

It turns out that we can represent the game much more concisely if we take advantage of the fact that many paths of play lead to the guesser knowing the exact same information. For example, if the player guesses GIANT and gets reply of 2 followed by guessing PECAN with a reply of 3, he knows the exact same information as if he had guessed PECAN with a reply of 3 followed by GIANT with a reply of 2; both sequences should lead to the same game state. More generally, two sequences of guesses and answers lead to identical knowledge bases if and only if the set of words consistent with both sequences is identical. Thus, there is a one-to-one correspondence between a knowledge base of the guesser and a subset of the set of words in the dictionary corresponding to the set of words that are consistent with the guesses so far. Since there are 2^{2833}

[2] If this restriction were not imposed, then players might need to guess all possible permutations of a word even once all the letters are known.

total subsets of words in the dictionary, the total number of game states where the guesser would need to make a decision is $2^{2833} \approx 10^{853}$. Since the best equilibrium-finding algorithms can only solve games with up to 10^{12} nodes, we have no hope of solving Jotto by simply applying an existing algorithm.

Furthermore, Jotto does not seem amenable to the same abstraction paradigm that has been successful on poker. In poker, abstraction works by grouping several states into the same *bucket* and forcing all states in the same bucket to follow the same strategy. In our compact representation of Jotto, the states correspond to subsets of the dictionary; abstraction would mean that several subsets are grouped together into single buckets. However, the action taken at each bucket will be a single word (i.e., the next guess). If many subsets are grouped together into the same bucket, then there will clearly be some words that have already been guessed in some states in the bucket, while not in other states. It will lead to certain actions being ill-defined, as well as possible infinite loops in the structure of the abstracted game tree. It will be exacerbated by the fact that many states will need to be grouped into the same bucket; on average, each bucket will contain 10^{841} game states.

In short, there are significant challenges that must be overcome to apply any sort of abstraction to Jotto; this may not even be feasible to do at all. Instead, we propose an entirely new approach.

3 A Natural Approach

A natural strategy for the hider would be to select each word uniformly at random, and one for the guesser would be to guess always the word that will eliminate the most words that are still consistent with the guesses and answers so far (in expectation against the uniform hider strategy). We refer to this strategy for the hider as HiderUniform, and to the strategy for the guesser as GuesserGBRUniform (for "Greedy Best Response"). GuesserGBRUniform is essentially 1-ply minimax search; further details and pseudocode are given in Section 7.1. We suspect that current Jotto programs follow algorithms similar to HiderUniform and GuesserGBRUniform, though we are not aware of any publicly available descriptions of existing algorithms. We will use these algorithms as a benchmark to measure the performance of our new approach.

While HiderUniform seems like a pretty strong (i.e., low-exploitability) strategy for the hider, it turns out that GuesserGBRUniform is actually highly exploitable. For example, in the five-letter variant using our rules and dictionary, GuesserGBRUniform will always select 'doyen' as its first guess. Clearly no intelligent hider would select doyen as his secret word against such an opponent. Furthermore, the hider can always guarantee that GuesserGBRUniform will require 9 guesses by selecting a word such as 'amped' (note that 9 is the maximal number of guesses that GuesserGBRUniform will take to guess any word in the 5-letter variant).

Searching additional levels down the tree will probably not help much with the worst-case exploitability of the guesser's strategy. The main problem is that GuesserGBRUniform plays a deterministic strategy, and a worst-case opponent

could exploit it by always selecting the word that requires the most guesses. We would like to compute a less exploitable strategy for the guesser, which will involve some amount of randomization. Our overall goal is to compute strategies for both players with worst-case exploitabilities as low as possible (i.e., we would like to compute an approximate Nash equilibrium, viewing Jotto as a game of imperfect information).

4 Game Theory Background

In this section, we review relevant definitions and prior results from game theory and game solving.

4.1 Strategic-Form Games

The most basic game representation, and the standard representation for simultaneous-move games, is the *strategic form*. A *strategic-form game* (aka matrix game) consists of a finite set of players N, a space of *pure strategies* S_i for each player, and a utility function $u_i : \times_i S_i \rightarrow \mathbb{R}$ for each player. Here $\times_i S_i$ denotes the space of *strategy profiles* — vectors of pure strategies, one for each player.

The set of *mixed strategies* of player i is the space of probability distributions over his pure strategy space S_i. We will denote this space by Σ_i. If the sum of the payoffs of all players equals zero at every strategy profile, then the game is called *zero sum*. In this paper, we will be primarily concerned with two-player zero-sum games. If the players are following strategy profile σ, we let σ_{-i} denote the strategy taken by the opponent.

4.2 Extensive-Form Games

An *extensive-form* game is a general model of multiagent decision-making with potentially sequential and simultaneous actions and imperfect information. As with perfect-information games, extensive-form games consist primarily of a game tree; each non-terminal node has an associated player (possibly *chance*) that makes the decision at that node, and each terminal node has associated utilities for the players. Additionally, game states are partitioned into *information sets*, where the player whose turn it is to move cannot distinguish among the states in the same information set. Therefore, in any given information set, the player whose turn it is to move must choose actions with the same distribution at each state contained in the information set. If no player forgets information that he previously knew, we say that the game has *perfect recall*.

4.3 Mixed vs. Behavioral Strategies

There are multiple ways of representing strategies in extensive-form games. Define a *pure strategy* to be a vector that specifies one action at each information set. Clearly there will be an exponentially number of pure strategies in the size

of the game tree; if the tree has I_i information sets for player i and A_i possible actions at each information set, then player i has $A_i^{I_i}$ possible pure strategies. Define a *mixed strategy* to be a probability distribution over the space of pure strategies. We can represent a mixed strategy as a vector with $A_i^{I_i}$ components.

Alternatively, we could play a strategy that randomizes independently at each information set; we refer to such a strategy as a *behavioral strategy*. Since a behavioral strategy must specify a probability for playing each of A_i actions at each information set, we can represent it as a vector with only $A_i \cdot I_i$ components. Thus, behavioral strategies can be represented exponentially more compactly than mixed strategies.

Providentially, it turns out that this gain in representation size does not come at the loss of expressiveness; any mixed strategy can also be represented as an equivalent behavioral strategy (and vice versa). Thus, current computational approaches to extensive-form games operate on behavioral strategies and avoid the unnecessary exponential blowup associated with using mixed strategies.

4.4 Nash Equilibria

Player i's *best response* to σ_{-i} is any strategy in $\arg\max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i})$. A *Nash equilibrium* is a strategy profile σ such that σ_i is a best response to σ_{-i} for all i. An ϵ-*equilibrium* is a strategy profile in which each player achieves a payoff of within ϵ of his best response.

In two player zero-sum games, we have the following result which is known as the *minimax theorem*:

$$v^* = \max_{\sigma_1 \in \Sigma_1} \min_{\sigma_2 \in \Sigma_2} u_1(\sigma_1, \sigma_2) = \min_{\sigma_2 \in \Sigma_2} \max_{\sigma_1 \in \Sigma_1} u_1(\sigma_1, \sigma_2).$$

We refer to v^* as the *value* of the game to player 1. Any equilibrium strategy for a player guarantees an expected payoff of at least the value of the game to that player.

All finite games have at least one Nash equilibrium. Currently, the best algorithms for computing a Nash equilibrium in two-player zero-sum extensive-form games with perfect recall are able to solve games with 10^{12} states in their game tree [7].

5 Smoothed Fictitious Play

In this section we will review the fictitious play (FP) algorithm [3]. Despite its conceptual simplicity, FP has recently been used to compute equilibria in many classes of games in the artificial intelligence literature (e.g., [4,5]). The basic FP algorithm works as follows. At each iteration, each player plays a best response to the average strategy of his opponent so far (we assume the game has two players). Formally, in smoothed fictitious play each player i applies the following update rule at each time step t:

$$s_{i,t} = \left(1 - \frac{1}{t+1}\right) s_{i,t-1} + \frac{1}{t+1} s_{i,t}^{BR},$$

where $s_{i,t}^{BR}$ is a best response of player i to the strategy $s_{-i,t-1}$ of his opponent at time $t-1$. We allow strategies to be initialized arbitrarily at $t=0$.

FP is guaranteed to converge to a Nash equilibrium in two-player zero-sum games; however, very little is known about how many iterations are needed to obtain convergence. Recent work shows that FP may require exponentially many iterations in the worst case [2]; however, it may perform far better in practice on specific games. In addition, the performance of FP is not monotonic; for example, it is possible that the strategy profile after 200 iterations is actually significantly closer to equilibrium than the profile after 300 iterations. So simply running FP for some number of iterations and using the final strategy profile is not necessarily the best approach.

We instead use the following improved algorithm. For each iteration we compute the amount each player could gain by deviating to a best response; denote it by $\epsilon_{i,t}$. Let $\epsilon_t = \max_i \epsilon_{i,t}$, and let $\epsilon_{t^*} = \min_{0 \le t \le T} \epsilon_t$. After running FP for T iterations, rather than output $s_{i,T}$, we will instead output s_{i,t^*} — the ϵ-equilibrium for smallest ϵ out of all all the iterations of FP so far.

6 Oracular Strategies

Consider the following scenario. Assume one is playing an extensive-form game G with 2^{100} information sets and two actions per information set, and that he wishes to play an extremely simple strategy: always choose the first action at each information set (assume actions are labeled as Action 1 and Action 2). To represent this pure strategy, technically we must list out a vector of size 2^{100} (with each entry being a 1 for this particular strategy). In contrast, it is trivial to write an algorithm that takes as input the index of an information set and outputs the action taken by this strategy (i.e., output 1 on all inputs). Even though there are a large number of information sets, we only require 100 bits to represent the index of each one; thus, it is possible to play this simple strategy without ever explicitly representing it.

More generally, let O_i be an efficient deterministic algorithm that takes as input the index of an information set I and outputs an action from A_I, the set of actions available at I. We refer to O_i as a *pure oracular strategy* for player i. It is easy to see that every pure oracular strategy is strategically equivalent to a pure strategy s_i of the game; at information set I, s_i straightforwardly plays whatever action O_i outputs on input I.

We define oracular versions of randomized strategies analogously to the extensive-form case. Let $\{O_i\}$ be a collection of pure oracular strategies; then any probability distribution over elements of $\{O_i\}$ is a *mixed oracular strategy.* If we let O_i be a randomized algorithm that outputs a probability distribution over actions at each information set, then call O_i a *behavioral oracular strategy.* As was the case with pure strategies, each oracular strategy corresponds to a single extensive-form strategy of the same type.

In the next two sections, we will see how the oracular strategy representation can be useful in practice when computing approximate equilibrium strategies in

Jotto. In particular, a strategy for the guesser is so large that we cannot represent it explicitly; however, we can encode it concisely as an oracular strategy which we efficiently query repeatedly throughout the algorithm.

7 Computing Best Responses in Jotto

In order to apply smoothed fictitious play to Jotto, we must figure out how to compute a best response for each player. This is challenging for several reasons. First, the guesser's strategy space is so large that we cannot compute a full best response; we must settle for computing an approximate best response, which we call the guesser's *greedy best response*. In addition, we represent the guesser's strategy in oracular form; so the hider cannot operate on it explicitly, and can only query it at certain game states. It turns out that we can actually compute an exact best response for the hider despite this limitation.

7.1 Computing the Guesser's Greedy Best Response

Assume we are given a strategy for the hider, and wish to compute a counter-strategy for the guesser. Let h denote the strategy of the hider, where h_i denotes the probability that the hider chooses w_i — the i'th word in the dictionary. Let D denote the number of words in the dictionary, and let S be a bit-vector of size D, where $S_i = 1$ means that w_i is still consistent with the guesses so far. So S encodes the current knowledge base of the guesser and represents the state of the game.

A reasonable heuristic to use for the guesser would be the following. For each word w_i in the dictionary, compute the number of words that we will eliminate in expectation (over h) if we guess w_i. Then guess the word that expects to eliminate the most words. We refer to this algorithm as GuesserGBR (for "Greedy Best Response"); pseudocode is given in Algorithm 1.

GuesserGBR relies on a number of subroutines. ExpNumElims, given in Algorithm 2, gives the expected number of words eliminated if w_i is guessed. AnswerProbs, given in Algorithm 4, gives a vector of the expected probability of receiving each answer from the hider when w is guessed. NumElims, given in Algorithm 3, gives the number of words that can be eliminated when w_i is guessed and an answer of j is given. Finally, NumCommLetts, given in Algorithm 5, gives the number of common letters between two words. In the pseudocode, L denotes the number of letters allowed per word. For efficiency, we precompute a table of size D^2 storing all the numbers of common letters between pairs of words. The overall running time of GuesserGBR is $O(D^2L)$.

It is worth noting that the greedy best response is not an actual best response; it is akin to searching one level down the game tree and then using the evaluation function "expected number of words eliminated" to determine the next move. This is essentially 1-ply minimax search. While we would like to compute an exact best response by searching the entire game tree, this is not feasible since

Algorithm 1. GuesserGBR(h, S)

```
for i = 1 to D do
    n_i ← ExpNumElims(w_i, h, S)
end for
return w_i with maximal value of n_i
```

Algorithm 2. ExpNumElims(w_i, h, S)

```
A ← AnswerProbs(w_i, h, S)
n ← 0
for j = 1 to L do
    n ← n + A_j · NumElims(w_i, S, j)
end for
return n
```

Algorithm 3. NumElims(w_i, S, j)

```
counter ← 0
for k = 1 to D do
    if S_k = 1 and
    NumCommLetts(w_i, w_k) ≠ j then
        counter ← counter + 1
    end if
end for
return counter
```

Algorithm 4. AnswerProbs(w, h, S)

```
for i = 1 to L do
    A_i ← 0
end for
for i = 1 to D do
    if S_i = 1 then
        k ← NumCommLetts(w, w_i)
        A_k ← A_k + h_i
    end if
end for
Normalize A so its elements sum to 1.
return A
```

Algorithm 5. NumCommLetts(w_i, w_j)

```
counter ← 0
for m = 1 to L do
    c_1 ← mth character of w_i
    for n = 1 to L do
        c_2 ← nth character of w_j
        if c_1 = c_2 then
            counter ← counter + 1
        end if
    end for
end for
return counter
```

the tree has 10^{853} nodes. As with computer chess programs, we will need to settle for searching down the tree as far as we can, then applying a reasonable evaluation function.

7.2 Computing the Hider's Best Response

In order to compute the hider's best response (in the context of fictitious play), we will find it useful to introduce two data structures. Let IterNumGuesses (ING) and AvgNumGuesses (ANG) be two arrays of size D. The i'th component of ING will denote the number of guesses needed for the guesser's greedy best response to guess w_i at the current iteration of the algorithm. The i'th component of ANG will be the average over all iterations (of fictitious play) of the number of guesses needed for the guesser's greedy best response to guess w_i. We update ANG by applying

$$ANG[i] = \left(1 - \frac{1}{t+1}\right) ANG[i] + \frac{1}{t+1} ING[i].$$

We update ING at each time step by applying

$$ING = CompNumGuesses(s_{h,t}),$$

where $s_{h,t}$ is the hider's strategy at iteration t of fictitious play, and pseudocode for CompNumGuesses is given below in Algorithm 6. CompNumGuesses computes the number of guesses needed for the guesser's greedy best response to $s_{h,t}$ to guess correctly each word. It accomplishes this by repeatedly querying GuesserGBR at various game states. The subroutine UpdateState updates the

Algorithm 6. CompNumGuesses(s_h)

```
for i = 1 to D do
    S ← vector of size D of all ones.
    numguesses ← 0
    while TRUE do
        numguesses ← numguesses + 1
        nextguess ← GuesserGBR(s_h, S)
        answer ←
        NumCommLetts(w_i, nextguess)
        if answer = L then
            BREAK
        end if
        S ←
        UpdateState(S, nextguess, answer)
    end while
    x_i ← numguesses
end for
return x
```

Algorithm 7. UpdateState(S, *nextguess*, *answer*)

```
for i = 1 to D do
    if S_i = 1 and NumCommLetts(nextguess,
    w_i) ≠ answer then
        S_i ← 0
    end if
end for
return S
```

Algorithm 8. HiderBR(ANG)

```
x* ← max_i ANG[i].
T* ← {i : ANG[i] = x*}
for i = 1 to D do
    if ANG[i] = x* then
        y_i = 1/|T*|
    else
        y_i = 0
    end if
end for
return y
```

game state in light of the answer received from GuesserGBR. It is in this way that the hider's best response algorithm selectively queries the guesser's strategy, which is represented implicitly in oracular form.

Finally, once ING and ANG have been updated as described above, we are ready to compute the full best response for the hider. Pseudocode for the algorithm HiderBR is given below in Algorithm 8. HiderBR takes ANG as input, and determines which word(s) required the most guesses on average. If there is a unique word requiring the maximal number of guesses, then that word is selected with probability 1. If there are multiple words requiring the maximal number of guesses, then these are each selected with equal probability. Note that selecting any distribution over these words would constitute a best response; we just choose one such distribution. The asymptotic running time of CompNumGuesses is $O(D^4L)$, while that of HiderBR is $O(D)$.

Note that, unlike GuesserGBR of Section 7.1 which is an approximate best response using 1-ply minimax search, HiderBR is a full best response. We are able to compute a full best response for the hider because his strategy space is much smaller than that of the guesser; the hider has only D possible pure strategies — 2833 in the case of 5-letter Jotto.

7.3 Parallelizing the Best Response Calculation

The hider's best response calculation can be sped up drastically by parallelizing over several cores. In particular, we parallelize the CompNumGuesses subroutine as follows. For the first processor, we iterate over $i = 1$ to $\left\lfloor \frac{D}{P} \right\rfloor$, where P is the number of processors, and so on for the other processors. Thus we can perform P independent computations in parallel. The overall running time of the new algorithm is $O\left(\frac{D^4L}{P}\right)$.

8 Computing an Approximate Equilibrium in Jotto

We would like to apply smoothed fictitious play to Jotto, using HiderBR for the hider's best response and GuesserGBR for the guesser's best response; however, this is tricky for several reasons. We mention three of them. First, it is not clear how to compute the epsilons and determine the quality of our strategies. Second, it will be difficult to run the algorithm without explicitly represent the guesser's strategy. Third, we cannot output it at the end of the algorithm.

It turns out that using the data structures developed in Section 7.2, we can actually compute the epsilons quite easily. This is accomplished using the procedures given in Algorithms 9–12.

We are now ready to present our full algorithm for computing an approximate equilibrium in Jotto; pseudocode is given in Algorithm 13. Note that we initialize the hider's strategy to choose each word uniformly at random. In terms of the guesser's strategy, it turns out that all the information needed to obtain it is already encoded in the hider's strategy and that we do not actually need to represent it in the course of the algorithm.

To obtain the guesser's final strategy, note that the strategies of the hider are output to a file at each iteration. It turns out that we can use this file to generate efficiently samples from the guesser's strategy, even though we never explicitly output this strategy. We present pseudocode in Algorithm 14 for generating a sample of the guesser's strategy at state S from the file output in Algorithm 13. This algorithm works by randomly selecting an integer t from 1 to t^*, then playing the guesser's greedy best response to $s_{h,t}$ — the hider's strategy at iteration t. We can view this algorithm as representing the guesser's strategy as a mixed oracular strategy; in particular, it is the uniform distribution over his greedy best responses in the first t^* iterations of Algorithm 13. This is noteworthy since it is a rare case of the mixed strategy representation having a computational advantage over the behavioral strategy representation.

9 Results

We ran our algorithm SolveJotto on four different Jotto instances, allowing words to be 2, 3, 4, or 5 letters long. To speed up the computation, we used the parallel version of the bottleneck subroutine CompNumGuesses (described in Section 7.3) with 16 processors. As our dictionary, we use the Official Tournament and Club Word List [6], the official word list for tournament Scrabble in several countries. As discussed in Section 2, we omit words with duplicate letters and words for which there exists an anagram that is also a word. The dictionary sizes are given in Table 1. We note that our algorithms extend naturally to any number of words and dictionary sizes (and to other variants of Jotto as well).

One metric for evaluating our algorithm is to play the strategies it computes against a benchmark algorithm. The benchmark algorithm we chose selects his word uniformly at random as the hider, and plays the greedy best response to the uniform strategy as the guesser. This is the same strategy that we use to initialize our algorithm.

Algorithm 9. HiderBRPayoff(ANG)

```
maxnumguesses ← 0
for i = 1 to D do
    if ANG[i] > maxnumguesses then
        maxnumguesses ← ANG[i]
    end if
end for
return maxnumguesses
```

Algorithm 10. HiderActualPayoff(ANG, s_h)

```
result ← 0
for i = 1 to D do
    result ← result + ANG[i] · s_h[i]
end for
return result
```

Algorithm 11. GuesserBRPayoff(ING, s_h)

```
result ← 0
for i = 1 to D do
    result ← result + ING[i] · s_h[i]
end for
return -1 · result
```

Algorithm 12. GuesserActualPayoff(ANG, s_h)

```
return -1 · HiderActualPayoff(ANG, s_h)
```

Algorithm 13. SolveJotto(T)

```
s_{h,0} ← (1/D, ..., 1/D)
Output s_{h,0} to StrategyFile
ING ← ComputeNumGuesses(s_{h,0})
ANG ← ING
Compute ε's per Algorithms 9-12, t* ← 0
for t = 1 to T do
    s^{BR}_{h,t} ← HiderBR(ANG)
    s_{h,t} ← (1 − 1/(t+1)) s_{h,t−1} + 1/(t+1) s^{BR}_{h,t}
    Output s_{h,t} to StrategyFile
    ING ← ComputeNumGuesses(s_{h,t})
    ANG ← (1 − 1/(t+1)) ANG + 1/(t+1) ING
    Compute ε's per Algorithms 9-12
    if ε < ε* then
        ε* ← ε, t* ← t
    end if
end for
return (s_{h,t*}, t*, StrategyFile)
```

Algorithm 14. ObtainGuesserStrategy(StrategyFile, t^*, S)

```
i ← randint(1, t*)
s_h ← strategy vector on i'th line of StrategyFile
return GuesserGBR(s_h, S)
```

For each number of letters, we computed the payoff of our algorithm SOLVEJOTTO against the benchmark (recall that the payoff to the hider is the expected number of guesses needed for the guesser to guess correctly the hider's word). The overall payoff is the average of the hider and guesser payoff.

Several observations from Table 1 are noteworthy. First, our algorithm beats the benchmark for all dictionary sizes. In the two-letter game, our expected payoff against the benchmark is 0.517; our strategy requires over a full guess less than the benchmark in expectation. Our profit against the benchmark decreases as more letters are used.

In addition to head-to-head performance against the benchmark, we also compared the algorithms in terms of worst-case performance. Recall that ϵ denotes the maximum payoff improvement one player could gain by deviating to a best response (full best response for the hider and greedy best response for the guesser). Note that in all cases, our ϵ is significantly lower than that of the benchmark. For example, in the two-letter game the benchmark obtains an ϵ of 5.373, while our algorithm obtains one of 0.038.

Interestingly, we also observe that the self-play payoff of our algorithm, which is an estimate of the value of the game, does not increase monotonically with the number of letters. That is, increasing the number of letters in the game does not necessarily make it more difficult for the guesser to guess the hider's word.

Table 1. Summary of our experimental results

Number of letters	2	3	4	5
Dictionary size	51	421	1373	2833
Our hider payoff vs. benchmark	7.652	7.912	7.507	7.221
Our guesser payoff vs. benchmark	-6.619	-7.635	-7.415	-7.216
Our overall payoff vs. benchmark	0.517	0.139	0.046	0.003
Benchmark self-play hider payoff	6.627	7.601	7.365	7.079
Our algorithm self-play hider payoff	7.438	7.658	7.390	7.162
Benchmark epsilon	5.373	3.399	1.635	1.921
Our final epsilon	0.038	0.334	0.336	0.335
Number of iterations	22212	10694	3568	3906
Avg time per iteration (minutes)	3.635×10^{-4}	0.028	1.160	12.576

10 Conclusion

We presented a new approach for computing approximate-equilibrium strategies in Jotto. Our algorithm produces strategies that significantly outperform a benchmark algorithm with respect to both head-to-head performance and worst-case exploitability. The algorithm extends fictitious play to a novel strategy representation called oracular form. We expect our algorithm and the oracular form representation to apply naturally to many other interesting games as well; in particular, games where the strategy space is very large for one player, but relatively small for the other player.

References

1. Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., Szafron, D.: Approximating game-theoretic optimal strategies for full-scale poker. In: IJCAI (2003)
2. Brandt, F., Fischer, F., Harrenstein, P.: On the Rate of Convergence of Fictitious Play. In: Kontogiannis, S., Koutsoupias, E., Spirakis, P.G. (eds.) SAGT 2010. LNCS, vol. 6386, pp. 102–113. Springer, Heidelberg (2010)
3. Brown, G.W.: Iterative solutions of games by fictitious play. In: Koopmans, T.C. (ed.) Activity Analysis of Production and Allocation (1951)
4. Ganzfried, S., Sandholm, T.: Computing an approximate jam/fold equilibrium for 3-player no-limit Texas Hold'em tournaments. In: AAMAS (2008)
5. Rabinovich, Z., Gerding, E., Polukarov, M., Jennings, N.R.: Generalised fictitious play for a continuum of anonymous players. In: IJCAI (2009)
6. Official tournament and club word list, http://www.isc.ro/en/commands/lists.html
7. Zinkevich, M., Bowling, M., Johanson, M., Piccione, C.: Regret minimization in games with incomplete information. In: NIPS (2007)

Online Sparse Bandit for Card Games

David L. St-Pierre[1], Quentin Louveaux[1], and Olivier Teytaud[2,3]

[1] Department of Electrical Engineering and Computer Science, Faculty of Engineering, Liège University, Belgium
[2] TAO (Inria, Lri, Univ. Paris-Sud, UMR CNRS 8623), France
[3] OASE Lab., National University of Tainan, Taiwan

Abstract. Finding an approximation of a Nash equilibrium in matrix games is an important topic that reaches beyond the strict application to matrix games. A bandit algorithm commonly used to approximate a Nash equilibrium is EXP3 [3]. However, the solution to many problems is often sparse, yet EXP3 inherently fails to exploit this property. To the best knowledge of the authors, there exist only an offline truncation proposed by [9] to handle such issue. In this paper, we propose a variation of EXP3 to exploit the fact that a solution is sparse by dynamically removing arms; the resulting algorithm empirically performs better than previous versions. We apply the resulting algorithm to an MCTS program for the Urban Rivals card game.

1 Introduction

Bandits algorithms [1,3,5,6] are tools for handling the dilemma of exploration vs exploitation. In particular, they are useful for finding Nash equilibria of matrix games [5,1]. Finding Nash equilibria of matrix games is an important topic; beyond its strict application, which models many financial applications, matrix games are a component of many games, even those represented by a set of rules. It has been shown in [9] how matrix games can be used inside a Monte-Carlo Tree Search implementation, for extending Monte-Carlo Tree Search to games with simultaneous actions, and [4] has extended this to games with partial information.

It is possible to use Linear Programming (LP) to compute an exact Nash Equilibrium (NE) of a zero-sum matrix game; however, it is not possible to apply LP for solving huge matrix games as players commonly encounter. Indeed, in many cases of interest (e.g., when used inside a Monte-Carlo Tree Search implementation), we want to minimize the number of accesses to the matrix, because the cost of solving the game lies more in the computation of the numbers in the matrix than in the solving itself. We want algorithms which find an approximate solution without reading the entire matrix. As [5] shows, one can find an ϵ-approximate Nash equilibrium of a zero-sum $K \times K$ matrix game by accessing only $O(K \log(K)/\epsilon^2)$ elements of the matrix, i.e., by far less than the $K \times K$ elements of the matrix. As shown in [1], EXP3 and INF, an EXP3-like

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 295–305, 2012.
© Springer-Verlag Berlin Heidelberg 2012

variant to address the adversarial multiarmed bandit problem, have the same property; moreover, they can be applied in a more general setting, namely, when each element of the matrix is only known up to a finite number of measurements, the average of which converges to the real value. As a consequence, bandit tools are now known as a standard solution for approximate solving of matrix games.

However, a crucial element in games has not been used in these works: the fact that, typically, the solutions found in games are sparse. In other words, a game can contain a large number of choices, yet only a very few among them are interesting.

This paper focuses on problems where the computation via LP of a NE is too intensive and therefore needs to be approximated. Furthermore, we seek an algorithm that does not require visiting each element of a potentially very large matrix. Based on this premise, to the best of our knowledge, the only paper in which sparsity in Nash equilibria was used for accelerating bandits is [9], with the offline algorithm discussed in section 2.3; we here investigate improved versions, working online. Section 2 presents the framework of sparse bandits and related algorithms. Section 3 presents experimental results on artificial and real world datasets.

2 Algorithms for Sparse Bandits

Sparse bandits are bandit problems in which the optimal solution is sparse; in the case of bandits for approximating Nash equilibria, this means that the Nash equilibria use only a small number of components, i.e., if x^*, y^* is a Nash equilibrium and the matrix has size $K \times K$, then $\{i; x^*_i > 0\}$ and $\{i; y^*_i > 0\}$ both have cardinal $<< K$.

We present below the framework of Nash equilibria in matrix games (section 2.1). We then present our version of the EXP3 algorithm (section 2.2), and thereafter our modified versions for sparse problems (section 2.3).

2.1 Matrix Games and Nash Equilibria

Consider a matrix M of size $K \times K$ with values in $[0, 1]$ (we choose a square matrix for short notations, but the extension is straightforward). Player 1 chooses an action $i \in [[1, K]]$ and player 2 chooses an action $j \in [[1, K]]$; both actions are chosen simultaneously. Then, player 1 gets reward $M_{i,j}$ and player 2 gets reward $1 - M_{i,j}$; the game therefore sums to 1. We consider games summing to 1 for commodity of notations in EXP3, but 0-sum games are obviously equivalent. A Nash equilibrium of the game is a pair (x^*, y^*) (both in $[0, 1]^K$ and summing to 1) such that if i is distributed according to the distribution x^* (i.e., $i = k$ with probability x^*_k) and if j is distributed according to the distribution y^* (i.e., $j = k$ with probability y^*_k) then neither player can expect a better average reward by unilaterally changing its strategy.

2.2 EXP3 Algorithm

We use a version of EXP3[3] inspired by [1]. EXP3 is an algorithm for bandits problems; here, we use two EXP3 bandits simultaneously: one for the row player and one for the column player. This is a tool for adversarial bandits, including matrix games.

Our version of EXP3 is explained in Alg. 1. Alg. 1 would be run for each player independently.

Algorithm 1. EXP3 algorithm for iteration t with K arms.

Initialise $\forall i, p(i) = \frac{1}{K}, n(i) = 0, S(i) = 0$; $t = 0$
while t < T **do**
  Arm i is chosen with probability $p(i)$
  n(i) ← n(i)+1
  Receive reward r
  t ← t+1
  S_i modified by the update formula $S_i \leftarrow S_i + r/p(i)$ (and S_j for $j \neq i$ is not modified).
  $\forall i, p(i) = 1/(K\sqrt{t}) + (1 - 1/\sqrt{t}) \times \exp(S_i/\sqrt{t})/\sum_j \exp(S_j/\sqrt{t})$
end while
return n

The ratio between the number of times an arm was pulled and the total number of iterations converges to the Nash equilibrium as explained in [2].

2.3 Sparse EXP3 Bandits

In the subsections below, we introduce a truncated EXP3 (A) and introduce our online pruning versions (B)

A: Truncated EXP3. [9] proposed to modify the EXP3 algorithm to exploit in a better way the sparsity in the solution, as explained in Alg. 2. The first step is to run the EXP3 algorithm, providing an approximation (x, y) of the Nash equilibrium and the second step executes a truncation following Alg. 2. This truncation uses the property that, over time, the probability to pull an arm that is not part of the optimal solution will tend toward 0. Therefore, as they are likely to be outside the optimal solution, it artificially truncates the arms which have a low probability to be pulled, based upon a threshold c.

The constant c is chosen as $\max_i(Tx_i)^\alpha/T$ for some $\alpha \in]0,1[$ (and d accordingly), as in [9], and T is the number of iterations of the EXP3 algorithm. $\alpha = 0.8$ is proposed in [9].

B: onEXP3. The question investigated in this paper is whether it is possible to truncate dynamically some arms to improve performance. In its current form, the online EXP3 algorithm that we propose assumes the total number of iterations

Algorithm 2. TEXP3, the truncation after EXP3.

Let x and y be the approximate Nash equilibria as proposed by EXP3 for the row and column players respectively.
Truncate as follows:

$$x'_i = x_i \text{ if } x_i > c, x'_i = 0 \text{ otherwise;}$$
$$y'_i = y_i \text{ if } y_i > d, y'_i = 0 \text{ otherwise.}$$

Renormalize: $x'' = x' / \sum_i x'_i$; $y'' = y' / \sum_i y'_i$.
Output x'', y''.

T is known. This assumption is relatively common in games as it can be viewed as an upper bound on the time allowed to take a decision.

Basically, onEXP3 extends TEXP3 by gradually increasing a threshold c_t. The first version of onEXP3 is solely based on the number of iterations t to determine c_t. With the knowledge of T, it is easy to extrapolate a fixed c_T and let c_t tend toward c_T.

It is important to bear in mind that it is the ratio between the number of times a specific arm x_i is pulled over the total number of iterations T that converges toward a Nash equilibrium, not the probability $p(i)$ given in EXP3. Therefore, the decision on whether to prune is based upon the number of times the arm was pulled. The main pitfall of using this method is to remove an arm that was not explored sufficiently. To prevent this situation, a lower threshold was included. Any arm pulled less than this threshold cannot be removed. In the current version, we used $x_i > \lceil \frac{t}{K} \rceil$.

The point is to remove every arm that, based on the current information, will not fulfill the final condition $x_i > c_T$ (and $y_j > d_T$). We used $x_i < (b_1 \times T^\delta \times \left(\frac{t}{T}\right)^\beta)$ where t is the current number of iterations, T is the maximum number of iterations, b_1 and $\beta \in \mathbb{R}$ and $\delta \in]-1, 0[$ (typically $\delta = \alpha - 1$ and $\beta = 3$). In the worst case scenario, i.e., if every arm were pruned, we turn back to normal EXP3. To sum up, Algorithm 3 presents the modified EXP3 (for one player; the same algorithm is applied for the other player).

onEXP3v2. Algorithm 3 is built upon the assumption that the matrices are square. For rectangular matrices, let us assume for simplicity that $k_1 > k_2$ (k_1 number of rows, k_2 number of columns). In onEXP3v2, we assume that the cut depends not on the number of iterations, but rather on the number of arms. Therefore, the number of arms must be included in the dynamic truncation condition. We chose to use $max_i(x_i)^\alpha$ as the truncating factor.

Thus, $x_i < (b_1 \times T^\delta \times \left(\frac{t}{T}\right)^\beta)$ becomes $x_i < (b_2 \times max_i(x_i)^\alpha \times \left(\frac{t}{T}\right)^\beta)$ where $b_2 \in \mathbb{R}$. To mitigate the relative difference in terms of the number of times an arm

Algorithm 3. onEXP3, an online EXP3 algorithm with a cut solely based on T.

Initialise $\forall i, p(i) = \frac{1}{K}, n(i) = 0, S(i) = 0$; $t = 0$
while t < T **do**
  Arm i is chosen with probability $p(i)$
  n(i) ← n(i)+1
  t ← t+1
  Receive reward r
  S_i modified by the update formula $S_i \leftarrow S_i + r/p(i)$
  $\forall i, p(i) = 1/(K\sqrt{t}) + (1 - 1/\sqrt{t}) \times \exp(S_i/\sqrt{t})/\sum_j \exp(S_j/\sqrt{t})$
  if $x_i > \lceil \frac{t}{K} \rceil$ **and** $x_i < (b_1 \times T^\delta \times \left(\frac{t}{T}\right)^\beta)$ **then**
    Remove arm i
  end if
  if every arm has been pruned **then**
    Use plain EXP3
  end if
  Renormalize: $p = p/\sum_i p(i)$
end while
Execute the truncation TEXP3 as presented in 2.3
return n

was pulled between the two players, a factor $\frac{k_1}{k_2}$ is added in front of the segment $\lceil \frac{t}{K} \rceil$. To sum up, Algorithm 4 presents the modified online EXP3 (onEXP3v2).

3 Computational Results

This section presents the computational results. Section 3.1 shows the results on randomly generated deterministic matrix games. Section 3.2 presents a version on non-square matrix games. Section 3.3 describes the results on Urban Rival, a simultaneous stochastic rectangular matrix game.

To generate a matrix that gives a sparse Nash equilibrium, we create a method that takes the number of arms k and a sparsity parameter γ. Then, a square submatrix of size k' is filled with $\{0, 1\}$ uniformly distributed with equal probabilities, where $k' = k(1 - \gamma)$ represents a small number that depends on γ. The $k - k'$ rows are filled with 0 with a probability of $1 - \gamma$ and 1 with a probability of γ (and the opposite for the columns). To ensure that the algorithms do not exploit a form of smoothness in the solution, we shuffle the rows and the columns. This protocol is trivially extendable to rectangular matrices.

Table 1 and Table 2 show the results of games in different settings. The column 'vs' displays the opponents. The column 'γ' gives information on the relative sparsity of the solution the higher the number, the sparser the solution is. Empirical experiments showed that at 0.8 one can expect $0.3k$ arms in the solution, at 0.9 around $0.1k$ arms and at 0.99 approximately $0.05k$ arms. The columns starting with '$\frac{T}{k1 \times k2}$' followed by a number state the relative number of iterations executed to reach these results, given in percentage.

Algorithm 4. onEXP3v2, an online EXP3 algorithm with a cut based on k.

Initialise $\forall i, p(i) = \frac{1}{K}, n(i) = 0, S(i) = 0; t = 0$
while t <T **do**
  Arm i is chosen with probability $p(i)$
  n(i) ← n(i)+1
  t ← t+1
  Receive reward r
  S_i modified by the update formula $S_i \leftarrow S_i + r/p(i)$
  if $x_i > \frac{k_1}{k_2} \times \lceil \frac{t}{K} \rceil$ **and** $x_i < (b_2 \times max_i(x_i)^{\alpha} \times \left(\frac{t}{T}\right)^{\beta})$ **then**
    remove arm i
  end if
  $\forall i$ not discarded, $p(i) = 1/(K\sqrt{t}) + (1 - 1/\sqrt{t}) \times \exp(S_i/\sqrt{t})/\sum_j \exp(S_j/\sqrt{t})$
  Renormalize: $p = p/\sum_i p(i)$
end while
Execute the truncation TEXP3 as presented in 2.3
return n

The score is equal to the mean of its performance when it plays as the row player combined with when it plays as the column player. This measure is more reliable than the distance to the optimal solution because a near-optimal solution can be weak in general against other suboptimal players. Thus, when an algorithm is playing against the optimal solution it can never reach more than 0.5. Also, the Nash equilibrium is not always possible to compute in games, therefore a strong performance against other algorithms is at least as important as performing well against the best solution. In our case, we look for an algorithm that performs well against both Nash Equilibria and suboptimal players. Every score is given in percentage.

3.1 Generated Squared Matrix Games

Every experiment was conducted over 100 randomly generated matrices. The seed for the random selection was fixed to reduce variance (noise) in the outcomes.

In Table 1, onEXP3 clearly outperforms TEXP3 when $\frac{T}{k1 \times k2}$ is between 5% and 50% of the number of elements in the matrix when playing with 200 arms on each side. When the matrices have a size of 100 arms, results are still strong. A caveat, at 50 arms and a very low number of iterations, the performance is lower in only one setting 49% (at 0.05) in which onEXP3 is weaker than TEXP3. When the number of iterations increases (0.1, 0.25, 0.5), the score (53%, 52%, 54%) is well over 50%. This suggests a better performance from onEXP3 for greater numbers of arms. As for the sparsity (represented by γ), even though it makes the scores change from 51%, 53%, 62%, 56% at 0.8 to 57%, 53%, 56%, 53% at 0.99, it does not change the general domination of onEXP3 over TEXP3 for these settings. The general decrease in the score between 0.25 and 0.5 can be explained by the fact that TEXP3 makes better decisions as the amount of available information increases.

Table 1. Performance of onEXP3. $\beta = 3$ in all experiments.

vs	γ	$k_1 \times k_2$	$\frac{T}{k1 \times k2} = 5\%$	$\frac{T}{k1 \times k2} = 10\%$	$\frac{T}{k1 \times k2} = 25\%$	$\frac{T}{k1 \times k2} = 50\%$
onEXP3 vs TEXP3	0.9	200 × 200	54 ± 1.14	58 ± 1.03	57 ± 0.91	56 ± 2.42
onEXP3 vs TEXP3	0.9	100 × 100	54 ± 1.13	53 ± 0.91	54 ± 0.90	58 ± 2.37
onEXP3 vs TEXP3	0.8	100 × 100	51 ± 0.55	53 ± 1.01	62 ± 1.21	56 ± 2.24
onEXP3 vs TEXP3	0.99	100 × 100	57 ± 1.70	53 ± 0.82	56 ± 1.87	53 ± 1.99
onEXP3 vs TEXP3	0.9	50 × 50	49 ± 0.12	53 ± 1.02	52 ± 0.77	54 ± 1.99
onEXP3 vs EXP3	0.9	200 × 200	78 ± 1.08	80 ± 0.17	80 ± 0.34	80 ± 0.32
onEXP3 vs EXP3	0.9	100 × 100	76 ± 1.21	80 ± 0.22	79 ± 0.27	78 ± 0.39
onEXP3 vs EXP3	0.8	100 × 100	67 ± 0.87	68 ± 0.40	68 ± 0.30	69 ± 0.46
onEXP3 vs EXP3	0.99	100 × 100	88 ± 1.35	**92 ± 0.09**	90 ± 0.28	88 ± 0.37
onEXP3 vs EXP3	0.9	50 × 50	75 ± 1.16	80 ± 0.28	78 ± 0.50	78 ± 0.42
onEXP3 vs NE	0.9	200 × 200	29 ± 0.54	30 ± 0.53	37 ± 0.81	43 ± 0.65
onEXP3 vs NE	0.9	100 × 100	19 ± 0.66	31 ± 0.64	35 ± 0.60	40 ± 0.75
onEXP3 vs NE	0.8	100 × 100	28 ± 0.33	37 ± 0.38	38 ± 0.45	42 ± 0.55
onEXP3 vs NE	0.99	100 × 100	22 ± 1.67	38 ± 0.92	40 ± 0.96	42 ± 0.90
onEXP3 vs NE	0.9	50 × 50	18 ± 0.65	20 ± 0.79	33 ± 0.70	38 ± 0.82
TEXP3 vs NE	0.9	200 × 200	15 ± 0.19	17 ± 0.33	25 ± 0.91	38 ± 0.94
TEXP3 vs NE	0.9	100 × 100	17 ± 0.37	19 ± 0.47	28 ± 0.93	35 ± 0.95
TEXP3 vs NE	0.8	100 × 100	27 ± 0.20	28 ± 0.22	32 ± 0.57	38 ± 0.68
TEXP3 vs NE	0.99	100 × 100	19 ± 1.56	28 ± 1.61	38 ± 0.94	40 ± 0.98
TEXP3 vs NE	0.9	50 × 50	19 ± 0.65	19 ± 0.58	28 ± 0.99	35 ± 0.92

When onEXP3 plays against EXP3, the results are clear: onEXP3 performs much better. Obviously, since it is the purpose of the algorithm it is not surprising. The variation in the size of the matrices changes the relative score, yet it does not change the outcome: onEXP3 is stronger. The variation in the sparsity yields the same conclusion. The score is consistent with the previous finding.

onEXP3 vs NE and TEXP3 vs NE give an insight into their relative performance against the best possible player. Aside from the lowest arms setting (50 arms), onEXP3 is always higher than TEXP3, sometimes the difference being as much as 14% (31%-17% at 200 arms). From the result, we may infer that onEXP3 tends faster toward a good solution. However, because it prunes dynamically, it has no guarantee of convergence (albeit converges with most of our randomly generated matrices). It performs consistently better than TEXP3 against the Nash equilibrium.

Therefore, within the setting presented in Table 1, it appears that onEXP3 is better than TEXP3 against both an optimal player and a suboptimal one.

Overall, the more information each algorithm receives, the better it plays; From $\frac{T}{k1 \times k2}$ equal to 0.05 up to 0.5, each algorithm improved its play against the Nash equilibrium. It seems that at $\frac{T}{k1 \times k2} = 5$, onEXP3 does not have sufficient information to truncate thus explaining the relatively lower performance (still significantly better) against TEXP3 when compared to results with higher number of iterations. The best performances are with a higher number of arms which only means the tuning is more efficient for this size of matrices.

Let us look at the trends of performance between the size of the matrices. In the setting 50×50 the results, while not as strong as in 100×100, still show a better performance by onEXP3. When the matrices are 200×200, results are

Table 2. Performance of onEXP3v2 . $\beta = 3$ in all experiments.

vs	γ	$k_1 \times k_2$	$\frac{T}{k1 \times k2} = 5\%$	$\frac{T}{k1 \times k2} = 10\%$	$\frac{T}{k1 \times k2} = 25\%$	$\frac{T}{k1 \times k2} = 50\%$
onEXP3v2 vs TEXP3	0.9	400 × 40	52 ± 0.49	53 ± 0.37	55 ± 0.35	56 ± 0.33
onEXP3v2 vs TEXP3	0.9	200 × 20	51 ± 0.53	52 ± 0.51	52 ± 0.31	55 ± 0.29
onEXP3v2 vs TEXP3	0.9	100 × 10	52 ± 0.53	51 ± 0.54	54 ± 0.54	54 ± 0.39
onEXP3v2 vs TEXP3	0.8	200 × 20	51 ± 0.53	52 ± 0.54	55 ± 0.43	55 ± 0.34
onEXP3v2 vs TEXP3	0.99	200 × 20	51 ± 0.58	54 ± 0.63	52 ± 0.27	54 ± 0.29
onEXP3v2 vs TEXP3	0.9	200 × 50	52 ± 0.54	53 ± 0.37	54 ± 0.32	54 ± 0.25
onEXP3v2 vs EXP3	0.9	400 × 40	83 ± 0.28	84 ± 0.15	84 ± 0.14	83 ± 0.14
onEXP3v2 vs EXP3	0.9	200 × 20	67 ± 0.62	78 ± 0.42	84 ± 0.17	84 ± 0.16
onEXP3v2 vs EXP3	0.9	100 × 10	58 ± 0.62	62 ± 0.61	78 ± 0.50	83 ± 0.28
onEXP3v2 vs EXP3	0.8	200 × 20	61 ± 0.47	69 ± 0.40	74 ± 0.24	74 ± 0.20
onEXP3v2 vs EXP3	0.99	200 × 20	67 ± 0.83	87 ± 0.72	**94 ± 0.23**	93 ± 0.13
onEXP3v2 vs EXP3	0.9	200 × 50	79 ± 0.46	82 ± 0.23	82 ± 0.41	81 ± 0.13
onEXP3v2 vs NE	0.9	400 × 40	37 ± 0.39	39 ± 0.33	41 ± 0.27	42 ± 0.22
onEXP3v2 vs NE	0.9	200 × 20	26 ± 0.61	37 ± 0.42	43 ± 0.24	44 ± 0.19
onEXP3v2 vs NE	0.9	100 × 10	17 ± 0.69	22 ± 0.69	39 ± 0.58	45 ± 0.33
onEXP3v2 vs NE	0.8	200 × 20	31 ± 0.46	37 ± 0.42	42 ± 0.29	44 ± 0.23
onEXP3v2 vs NE	0.99	200 × 20	20 ± 0.86	41 ± 0.73	48 ± 0.23	48 ± 0.15
onEXP3v2 vs NE	0.9	200 × 50	33 ± 0.58	36 ± 0.48	37 ± 0.31	40 ± 0.31
TEXP3 vs NE	0.9	400 × 40	35 ± 0.37	38 ± 0.28	39 ± 0.24	41 ± 0.18
TEXP3 vs NE	0.9	200 × 20	25 ± 0.59	35 ± 0.45	42 ± 0.23	43 ± 0.17
TEXP3 vs NE	0.9	100 × 10	14 ± 0.66	20 ± 0.65	34 ± 0.57	42 ± 0.36
TEXP3 vs NE	0.8	200 × 20	30 ± 0.43	36 ± 0.40	40 ± 0.27	43 ± 0.20
TEXP3 vs NE	0.99	200 × 20	20 ± 0.79	37 ± 0.73	48 ± 0.19	48 ± 0.17
TEXP3 vs NE	0.9	200 × 50	33 ± 0.58	35 ± 0.23	37 ± 0.40	39 ± 0.29

impressive achieving a $92 \pm\ 0.09\%$ with few iterations ($\frac{T}{k1 \times k2} = 10$), suggesting that larger matrices yield a better performance.

When comparing onEXP3 and TEXP3 against the Nash equilibrium over a γ of 0.9 and 0.8, both improve their respective performance as sparsity gets lower. It is not really surprising considering that the real difficulty is to prune arms that are not promising. There is an observable peak in terms of performance of onEXP3 against TEXP3 in a high sparsity setting (0.99). onEXP3 truncates within the EXP3 iteration allowing to focus more rapidly on the few good arms thus explaining this peak.

3.2 Generated General Matrix Games

Each experiment was conducted over 500 randomly generated matrices. The seed for the random selection was fixed to reduce variance (noise) in the outcomes.

Table 2 presents the results achieved by the algorithms in a rectangular matrix setting. onEXP3v2 outperforms TEXP3 when $\frac{T}{k1 \times k2}$ is between 0.05 and 0.5 percent of the number of elements in the matrix when playing with 400×40 arms. When the matrices have a size of 200×20 arms, the results are still strong. At 100×10 arms the score is well over 50%. This suggests a better performance from onEXP3v2 when the number of arms augments. The sparsity has an impact on the score, yet does not change the general domination of onEXP3v2 over TEXP3 for these settings.

When onEXP3v2 is playing against EXP3, the results are clear: onEXP3v2 performs much better, with results similar to Table 1. Obviously, since the purpose of the algorithm is to lean faster towards a better solution, it is not surprising. The variation in the size of the matrices changes the relative score, yet it does not change the outcome: onEXP3v2 is stronger. The variation in the sparsity yields the same conclusion: the score is consistent with the previous finding.

The difference between the performance of onEXP3v2 and TEXP3 against the best possible player, while not as strong as in Table 1, still gives the edge to onEXP3v2. Again, it seems that larger matrices yield a better performance. onEXP3v2 statistically beats TEXP3 with a 95% confidence interval in most cases. Also, there was no setting in which the mean was under 50%. When onEXP3v2 plays against EXP3, the domination is statistically observable at an interval of 3 times the standard deviation, even managing a peak at $94 \pm 0.23\%$. At a lower sparsity (e.g., 0.8), onEXP3v2 does not perform as well as its predecessor onEXP3. Nevertheless, it still manages good performance overall, but the sparsity does not impact as much as in Table 1.

It seems the narrow side of the matrix limits the effectiveness of the sparsity pruning. A less extreme rectangular matrix, such as 200×50, demonstrates this conclusion by having values higher than 200×20 and lower than 200×200. A very interesting result is the impact of the amount of information on the score. The score does not change as drastically as it did in Table 1.

3.3 Application to UrbanRivals

For the sake of comparison, we applied onExp3v2 to the same card game as [9], namely Urban Rivals, thanks to the code kindly provided by the authors. We implemented onExp3 and made it play against the version of TEXP3 used in [9].

Urban Rivals (UR) is a widely played internet card game, with partial information. As pointed out in [9], UR can be consistently solved by a Monte-Carlo Tree Search algorithm (MCTS) thanks to the fact that the hidden information is frequently revealed: a sequence of time steps without revealed information are integrated into one single matrix game, so that the game is rephrased as a tree of matrix games with finite horizon. EXP3 is used in each node (therefore EXP3 is used for solving many matrix games), each reading of a coefficient in the payoff matrix at depth d of the tree being a simulated game (by MCTS itself) with depth $d-1$. As a consequence, reading coefficients in the payoff matrices at the root is quite expensive, and we have to solve the game approximately. We refer to [9] for more details on the solving of partial information game with periodically revealed information by MCTS with EXP3 bandit.

The overall algorithm is a Monte-Carlo Tree Search, with EXP3 bandits in nodes; the sparse bandit is applied at the root node of the algorithm. We compared results for a similar number of iterations and obtained success rates as presented in Table 3.

The main difficulty the algorithm faces in this application is the evaluation of the reward. When EXP3 is used in an MCTS settings, results are stochastic (in the sense that playing a same i and j for the row and column player respectively does not always lead to the same result). onExp3v2 relies on the assumption that the reward is fixed to execute a more aggressive pruning, which is not the case in this application. Yet with little tuning, onEXP3v2 significantly outperforms TEXP3 when the number of simulations reaches 1600 and 3200.

The drop in the score when the simulations reach 6400 and 12800 is consistent with the findings in Table 1 and Table 2. There is sufficient information gathered during the iteration phase to prune adequatly offline. As such, these values are in fact showing that onEXP3v2 remains as good as TEXP3 when the number of iterations is high.

Table 3. Results of onExp3 for UrbanRivals (against the version of TEXP3 developed by the authors of [9]). The improvement is moderate but significant; we point out that the game is intrinsically noisy, so that 55% is already a significant improvement as only the average of many games makes sense for evaluating the level of a player.

# simulations per move	score $\pm\sigma$
100	0.519 $\pm$ 0.014
200	0.509 $\pm$ 0.014
400	0.502 $\pm$ 0.014
800	0.494 $\pm$ 0.015
1600	0.558 $\pm$ 0.014
3200	0.549 $\pm$ 0.014
6400	0.494 $\pm$ 0.015
12800	0.504 $\pm$ 0.015

4 Conclusion

EXP3 and related algorithms are great tools for computing approximate Nash equilibria. However, they do not benefit from sparsity. There already exists versions of EXP3 for sparse cases, but these versions only benefit from an offline sparsity; EXP3 is run, and, thereafter, some arms are pruned. This paper proposes online pruning algorithms (onEXP3 and onEXP3v2 for square and rectangular matrices respectively), dynamically removing arms, with a variable benefit (from minor to huge depending on the framework), and in all cases a significantly non-negative result.

The benefit on the real-world game Urban Rivals is moderate but significant. Our online sparsity algorithm scales well as it becomes more and more efficient as the game becomes bigger.

The main further works are as follows. First, whereas TEXP3 (from [9]) makes no sense in internal nodes of a MCTS tree, our modified (online) version makes sense for all nodes - therefore, we might extend our application to Urban Rivals by applying online sparsity to all nodes, and not only at the root. This is a straightforward modification which might lead to big improvements.

The second further work is a formal analysis of onEXP3. Even TEXP3 only has a small theoretical analysis; the algorithms are so stable and so scalable that we believe a mathematical analysis is possible.

The third further work is to compare a framework based on the combination of MCTS with onEXP3 to other similar opponents such as MCCFR [7] and MCRNR [8].

Acknowledgements. This paper presents research results of the Belgian Network DYSCO (Dynamical Systems, Control, and Optimization), funded by the Interuniversity Attraction Poles Programme, initiated by the Belgian State, Science Policy Office. The scientific responsibility rests with its authors.

References

1. Audibert, J.Y., Bubeck, S.: Minimax policies for adversarial and stochastic bandits. In: 22nd Annual Conference on Learning Theory (COLT), Montreal (June 2009)
2. Audibert, J.-Y., Bubeck, S.: Regret bounds and minimax policies under partial monitoring. Journal of Machine Learning Research (October 2010)
3. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: Gambling in a rigged casino: the adversarial multi-armed bandit problem. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science, pp. 322–331. IEEE Computer Society Press, Los Alamitos (1995)
4. Auger, D.: Multiple Tree for Partially Observable Monte-Carlo Tree Search. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A.I., Merelo, J.J., Neri, F., Preuss, M., Richter, H., Togelius, J., Yannakakis, G.N. (eds.) EvoApplications 2011, Part I. LNCS, vol. 6624, pp. 53–62. Springer, Heidelberg (2011)
5. Grigoriadis, M.D., Khachiyan, L.G.: A sublinear-time randomized approximation algorithm for matrix games. Operations Research Letters 18(2), 53–58 (1995)
6. Lai, T., Robbins, H.: Asymptotically efficient adaptive allocation rules. Advances in Applied Mathematics 6(1), 4–22 (1985)
7. Lanctot, M., Waugh, K., Zinkevich, M., Bowling, M.: Monte Carlo Sampling for Regret Minimization in Extensive Games. Advances in Neural Information Processing Systems 22, 1078–1086 (2009)
8. Ponsen, M., Lanctot, M., de Jong, S.: MCRNR: Fast Computing of Restricted Nash Responses by Means of Sampling. In: Proceedings of Interactive Decision Theory and Game Theory Workshop, AAAI 2010 (2010)
9. Teytaud, O., Flory, S.: Upper Confidence Trees with Short Term Partial Information. In: Di Chio, C., Cagnoni, S., Cotta, C., Ebner, M., Ekárt, A., Esparcia-Alcázar, A.I., Merelo, J.J., Neri, F., Preuss, M., Richter, H., Togelius, J., Yannakakis, G.N. (eds.) EvoApplications 2011, Part I. LNCS, vol. 6624, pp. 153–162. Springer, Heidelberg (2011)

Game Tree Search with Adaptive Resolution

Hung-Jui Chang, Meng-Tsung Tsai, and Tsan-sheng Hsu*

Institute of Information Science, Academia Sinica, Taipei, Taiwan
{chj,mttsai,tshsu}@iis.sinica.edu.tw

Abstract. In this paper, we use an adaptive resolution R to enhance the min-max search with alpha-beta pruning technique, and show that the value returned by the modified algorithm, called Negascout-with-resolution, differs from that of the original version by at most R. Guidelines are given to explain how the resolution should be chosen to obtain the best possible outcome. Our experimental results demonstrate that Negascout-with-resolution yields a significant performance improvement over the original algorithm on the domains of random trees and real game trees in Chinese chess.

1 Introduction

In a 2-player perfect information game, the standard search technique is min-max search with alpha-beta pruning [10]. Several algorithms and heuristics based on the alpha-beta search algorithm have been proposed to improve the technique. Existing methods can be divided into four categories: (1) methods that adjust the size of the search interval so that good variants can be found in a tighter search interval, e.g., the NegaScout [14], Aspiration search [9], Probcut [8], Futility Pruning [7] and MTD(f) algorithms [12,13]; (2) methods that focus on finding a better move order by using techniques like the knowledge heuristic, refutation table, killer heuristic [2] or history heuristic [15]; (3) methods that try to increase the depth of the search by forward pruning, e.g., by using a transposition table [3], null move pruning [1], Multicut [4] or late move reduction [5]; and (4) methods that change the search depth dynamically based on the current game boards, e.g., quiescent search [6] and Rankcut [11]. Among the above approaches, finding a good move order, using a finely-tuned search interval, and aggressive forward pruning are the most widely used techniques.

A finely-tuned search interval enhances the performance of alpha-beta pruning by increasing the chances of performing a beta cut; thus, it provides an alternative play that may be as good as the original move. For example, in the Aspiration search technique with iterative deepening, the returned result of depth i can be used to estimate the expected result of depth $i+1$. If the returned value of depth i is s_i, the returned value of depth $i+1$ is estimated to be in the search interval $[s_i - w, s_i + w]$, where $w/2$ is the estimated window size. Instead of using $(-\infty, \infty)$ as in the alpha-beta search, Aspiration search uses $[s_i - w, s_i + w]$

* Corresponding author.

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 306–319, 2012.
© Springer-Verlag Berlin Heidelberg 2012

as the search interval; hence, there are three cases. In the first case, if the returned value of depth $i+1$, namely, s_{i+1}, is in the interval $[s_i - w, s_i + w]$, then the search behavior is similar to that of the original alpha-beta search method. In the second case, the alpha-beta search may fail because the value is too low, i.e., s_{i+1} is less than $s_i - w$, so the algorithm must re-search the game tree with the search interval $(-\infty, s_i - w]$. In the third case, the alpha-beta search may fail because the value is too high, i.e., s_{i+1} is greater than $s_i + w$. Hence, the algorithm does not need to re-search the game tree because the current returned value is good enough, i.e., greater than $s_i + w$. Moreover, the algorithm uses iterative deepening, so it can arrive at the exact value in the next search depth, i.e., depth $i+2$. The third case shows that, in certain positions, instead of finding the best move, we only need to find a move that is close to the best move.

In a 2-player game, the score of each move is determined by an evaluation function. Usually, the function is designed to consider several features in a game of chess, e.g., the material value of each piece, the mobility of each piece, and the king's safety [16]. In most cases, each feature has a distinct assigned weight and the evaluation function calculates the score by accumulating the product of each feature's value and the corresponding weight. However, when the scores of two moves only differ slightly, the move with the higher score does not need to have a higher score than the other move for every feature. Because the selection of each feature's weights is intuitive, scientific comparison of two scores when their difference is less 1% is not very meaningful. For example, on two different boards, if the mobility scores of a Rook are 10 and 8 respectively, and those of a Knight are 9 and 11 respectively, it is hard to decide whether the board with mobility of the Rook = 10 and the Knight = 9 is worse than the board with the mobility of the Rook = 8 and the Knight = 11. We also find that when we have a strong advantage and the score of a definite-win position is about 2000, there is little difference between playing a move with a returned score = 1000 and playing a move with a returned score = 1200. These observations suggest that searching for the theoretical best value may not be cost-effective if our objective is to win the game, rather than win it by the fastest possible means.

In this paper, we modify the alpha-beta pruning algorithm to (1) increase the chances of pruning more branches; and (2) address the problem of giving too much weight to the slight difference between moves caused by the design of the evaluation functions. We make a decision about each move with a *resolution factor* R, which depends on the current search score. Specifically, when the resolution value is equal to R, the original score s is treated as $\lfloor s/R \rfloor$. For example, in the original alpha-beta pruning algorithm, if the beta value is 70 and the value of one of the child nodes is 65, the child node would not be pruned because its value is smaller than the value of beta. However, if we set the resolution at 16, the value of the child node ($\lfloor 65/16 \rfloor = 4$) would be equal to the value of beta ($\lfloor 70/16 \rfloor = 4$), so the node would be pruned. By using a larger R, more branches can be pruned, but there is a greater risk of pruning the wrong branches. If R is equal to 1, the algorithm is equivalent to the original alpha-beta search. We carefully decide R dynamically to find a balance between speed and accuracy.

Some techniques also adjust the search parameter according to the current search state. For example, Probcut [8] assumes that a node's score can be estimated based on that of its predecessor. It then uses the estimated bound to increase the chances of making a cut. Meanwhile, Futility Pruning [7] uses the evaluation function's features to estimate whether a node has a chance of achieving a higher score than the current score. Although these techniques make a cut in a similar way to our approach, the motivation is different. They focus on estimating the range of the returned scores. By contrast, instead of estimating the scores, our method treats all the scores in one resolution equally.

The remainder of this paper is organized as follows. In Section 2, we define the notations used throughout the paper and present the algorithm; in Section 3, we demonstrate the correctness of the algorithm and analyze the error bound. We discuss the experimental environment, settings and results in Section 4. Then summarize our conclusions in Section 5.

2 Notations and Algorithm

Hereafter, we use $R \geq 1$ to represent the value of a resolution. Any real number s under a resolution R is treated as $s^R = \lfloor s/R \rfloor$; and $M(R, V)$ is the maximum function of an ordered set V under the resolution R. Function M compares every element in V with (1) the input order under resolution R, i.e., $M(R, V) = v_k$, where v_k has the largest value of $v_k^R = \lfloor v_k/R \rfloor$; and (2) the smallest index number k among all the elements that have the largest value v_k^R under resolution R.

Consider a game tree $T(V, E)$ comprised of a node set V and an edge set E, and let $B(u)$ be the children of a node u. In the original Negamax algorithm, the value of each node u is derived by

$$F(u) = \begin{cases} E(u) & \text{if } u \text{ is a leaf node of } T, \\ -\max_{v \in B(u)} F(v) & \text{otherwise,} \end{cases} \tag{1}$$

where $E(u)$ is the designed evaluation function. However, we modify Equation 1 by using the concept of resolution as follows:

$$F^R(u) = \begin{cases} E(u) & \text{if } u \text{ is a leaf node of } T, \\ -\max^R_{v \in B(u)} F^R(v) & \text{otherwise,} \end{cases} \tag{2}$$

where $\max^R_{v \in B(u)} F^R(v)$ is the maximum value of $F^R(v)$ returned by the maximum function M, i.e., $\max^R_{v \in B(u)} F^R(v) = M(R, V)$ with $V = \{F^R(v) \mid v \in B(u)\}$.

By applying Equation 2, we derive the Negascout-with-resolution procedure (Algorithm 1). The resolution scheme is used in Line 8 (value comparison) and Line 15 (beta cut). In Line 8, the value comparison step uses the scheme to compare the returned value in the previous line with the temporary maximum value. Then, in Line 15, the beta cut step uses the scheme to compare the current selected value with the beta value.

Algorithm 1. Negascout-with-resolution(position *p*, value *alpha*, value *beta*, integer *depth*, integer *R*)

Input: position p, value $alpha$, value $beta$, integer $depth$, integer R
Output: value m

```
1  if depth = 0 then
2  |  return E(p)
3  end
4  m:= −∞;
5  n:= beta;
6  foreach p_i ∈ B(p) do
7  |  t:= -Negascout-with-resolution(p_i,-n,-max{alpha, m}, depth − 1, R);
8  |  if ⌊t/R⌋ > ⌊m/R⌋ then /* apply the resolution scheme to
   |                            compare the values.              */
9  |  |  if n = beta then
10 |  |  |  m := t;
11 |  |  else
12 |  |  |  m := -Negascout-with-resolution(p_i,−beta,-t, depth − 1, R);
13 |  |  end
14 |  end
15 |  if ⌊m/R⌋ ≥ ⌊beta/R⌋ then /* apply the resolution scheme when
   |                              deciding a possible beta cut.    */
16 |  |  return m;
17 |  end
18 |  n := max{alpha, m} + 1;
19 end
20 return m;
```

Note that the resolution used in the beta cut step and the value comparison step can be set at different values. In Negascout-with-resolution, if we set both resolutions as 1, the algorithm behaves in the same way as the original Negascout algorithm. We claim that if we set both resolutions to be the same as R and Negascout-with-resolution returns a value v, then the optimal value is located in the interval $[R\lfloor v/R \rfloor, R\lfloor v/R \rfloor + R]$, i.e., the difference between the best value and the returned value v is at most R. We prove this claim in the next section.

3 Theoretical Analysis

In this section, we demonstrate that the difference between the values returned by the Negascout and Negascout-with-resolution algorithms is not more than the given resolution R. First, we prove a lemma that shows the difference between $M(R, V)$ and $M(1, V)$ is not more than R. Next, we prove that, for two ordered sets V and V^*, if the difference between each pair of elements with the same index number is not more than R, then the difference between $M(R, V)$ and

$M(R, V^*)$ will not more than R. Finally, we prove that, for a given game tree, the difference between the values returned by the original Negascout and Negascout-with-resolution is not more than the given resolution R.

First, we prove that, for a fixed R and ordered set V, $M(1, V)$ and $M(R, V)$ belong to the same interval $[R\lfloor M(R, V)/R\rfloor, R\lfloor M(R, V)/R\rfloor + R]$.

Lemma 1. *For any real value* $R \geq 1$ *and* $V = \{v_1, v_2, \ldots, v_n \mid v_i \in \mathbb{R}\}$, $R\lfloor M(R, V)/R\rfloor \leq M(R, V) < R\lfloor M(R, V)/R\rfloor + R$.

Proof. Since $\lfloor M(R, V)/R\rfloor \leq M(R, V)/R < \lfloor M(R, V)/R\rfloor + 1$ holds for all $R \geq 1$, multiplying both sides by R, we have $R\lfloor M(R, V)/R\rfloor \leq M(R, V) < R\lfloor M(R, V)/R\rfloor + R$. □

Lemma 2. *For any real value* $R \geq 1$ *and* $V = \{v_1, v_2, \ldots, v_n \mid v_i \in \mathbb{R}\}$, $R\lfloor M(R, V)/R\rfloor \leq M(1, V) < R\lfloor M(R, V)/R\rfloor + R$.

Proof. Since $M(1, V) = \max_{v_i \in V} v_i$ is the maximum value of V, $M(1, V) \geq M(R, V)$. For a fixed R, we consider the following two cases.

Case 1: if $M(1, V) = M(R, V)$, then $R\lfloor M(R, V)/R\rfloor \leq M(1, V) = M(R, V) < R\lfloor M(R, V)/R\rfloor + R$ by Lemma 1.

Case 2: if $M(1, V) > M(R, V)$, we assume that $v_i = M(1, V)$ and $v_j = M(R, V)$. Then, $\lfloor M(1, V)/R\rfloor = \lfloor v_i/R\rfloor \leq \lfloor v_j/R\rfloor = \lfloor M(R, V)/R\rfloor$ according to the definition of $M(R, v)$. Since $M(1, V) > M(R, V)$ and $\lfloor M(1, V)/R\rfloor \leq \lfloor M(R, V)/R\rfloor$, we have $\lfloor M(1, V)/R\rfloor = \lfloor M(R, V)/R\rfloor$, i.e., $M(1, V)/R < \lfloor M(1, V)/R\rfloor + 1 = \lfloor M(R, V)/R\rfloor + 1$. Multiplying both sides of the previous equation by R and applying the assumption that $M(1, V) > M(R, V)$ and Lemma 1, we have $R\lfloor M(R, V)/R\rfloor \leq M(R, V) < M(1, V) < R\lfloor M(R, V)/R\rfloor + R$.

The above cases demonstrate that Lemma 2 holds. □

Next, using Lemma 1 and Lemma 2, we show that the difference between $M(R, V)$ and $M(1, V)$ is not more than R.

Lemma 3. *For any real value* $R \geq 1$ *and* $V = \{v_1, v_2, \ldots, v_n \mid v_i \in \mathbb{R}\}$, $R\lfloor M(R, V)/R\rfloor \leq M(R, V), M(1, V) < R\lfloor M(R, V)/R\rfloor + R$, *i.e.,* $|M(R, V) - M(1, V)| < R$.

Proof. By Lemma 1 we have

$$R\lfloor M(R, V)/R\rfloor \leq M(R, V) < R\lfloor M(R, V)/R\rfloor + R. \tag{3}$$

By Lemma 2 we have

$$R\lfloor M(R, V)/R\rfloor \leq M(1, V) < R\lfloor M(R, V)/R\rfloor + R. \tag{4}$$

By subtracting Equation 4 from Equation 3 we have

$$-R < M(R, V) - M(1, V) < R, \tag{5}$$

i.e., $|M(R, V) - M(1, V)| < R$. □

Now we can show that, for two input ordered sets V and V^*, if the values of each pair of elements in the same index of the two ordered sets, e.g., v_i and v_i^*, are in an interval of length R, then the difference between the values returned by $M(R, V)$ and $M(R, V^*)$ is not more than R.

Lemma 4. *For any real value $R \geq 1$ and ordered sets $V = \{v_1, v_2, \ldots, v_n \mid v_i \in \mathbb{R}\}$ and $V^* = \{v_1^*, v_2^*, \ldots, v_n^* \mid \lfloor v_i/R \rfloor \leq v_i^*/R < \lfloor v_i/R \rfloor + 1\}$, $\lfloor M(R, V)/R \rfloor \leq M(R, V^*)/R < \lfloor M(R, V)/R \rfloor + 1$, i.e., $|M(R, V) - M(R, V^*)| < R$.*

Proof. Assume that $v_i = M(R, V)$ and $v_j^* = M(R, V^*)$. If v_i and v_j^* have the same index number, i.e., $v_j^* = v_i^*$, then, by the definition of v_i and v_J^*, we have $\lfloor M(R, V)/R \rfloor = \lfloor v_i/R \rfloor \leq v_i^*/R = M(R, V^*)/R < \lfloor v_i/R \rfloor + 1 = \lfloor M(R, V)/R \rfloor + 1$.

If v_i and v_j^* have different index numbers, we consider two cases; $v_i > v_j^*$ and $v_i \leq v_j^*$. In case 1, by the definition of v_i and v_j^*, $\lfloor M(R, V)/R \rfloor + 1 = \lfloor v_i/R \rfloor + 1 > v_i/R > v_j^*/R = M(R, V^*)/R \geq v_i^*/R \geq \lfloor v_i^*/R \rfloor = \lfloor v_i/R \rfloor = \lfloor M(R, V)/R \rfloor$. Similarly, in the second case, we have $\lfloor M(R, V)/R \rfloor + 1 = \lfloor v_j/R \rfloor + 1 > v_j^*/R = M(R, V^*)/R \geq v_i/R > \lfloor v_i/R \rfloor = \lfloor M(R, V)/R \rfloor$.

Then, based on the above discussion and Lemma 1, $\lfloor M(R, V)/R \rfloor \leq M(R, V^*)/R < \lfloor M(R, V)/R \rfloor + 1$ if $V^* = \{v_1^*, v_2^*, \ldots, v_n^* \mid \lfloor v_i/R \rfloor \leq v_i^*/R < \lfloor v_i/R \rfloor + 1\}$. Hence, we have $|M(R, V) - M(R, V^*)| < R$. □

Finally, we prove that the difference between the values returned by the original Negascout and Negascout-with-resolution algorithms is not more than R, where R is the given resolution.

Theorem 1. *For a given game tree with root p_r, the difference between the value returned by Negascout, $F(p_r)$, and the value returned by Negascout-with-resolution, $F^R(p_r)$, is not more than the given resolution R, i.e., $\lfloor F(p_r)/R \rfloor \leq F(p_r)/R, F^R(p_r)/R < \lfloor F(p_r)/R \rfloor + 1$ and $|F(p_r) - F^R(p_r)| < R$.*

Proof. We prove the theorem by induction.

First, we assume that the height of the game tree T is 1. Let p_r be the root of T. The returned values $F^R(p_r)$ and $F(p_r)$ are equal to $M(R, V)$ and $M(1, V)$ respectively, where V is the set of scores of each leaf, i.e., v_i is the corresponding value of leaf node $l_i \in B(p_r)$. According to Lemma 3, we have $\lfloor F(p_r/R \rfloor \leq F(p_r), F^R(p_r) < \lfloor F(p_r) \rfloor + 1$, i.e., $|F^R(p_r) - F(p_r)| < R$.

We assume that the statement also holds when the height of the game tree is h. Next, we consider a game tree T with a height of $h + 1$ rooted at node p_r. According to the induction hypothesis, for each node $p_i \in B(p_r)$, the score of p_i, $F^R(p_i)$, differs from $F(p_i)$ by at most R. For the ordered set $V = \{F(p_i) \mid p_i \in B(p_r)\}$, we have a corresponding ordered set $V^* = \{F^R(p_i) \mid p_i \in B(p_r)\}$ such that, for every $v_i^* \in V^*$, $\lfloor v_i/R \rfloor \leq v_i^*/R < \lfloor v_i/R \rfloor + 1$. According to Lemma 3 and Lemma 4, we have

$$\lfloor M(R, V)/R \rfloor \leq M(1, V)/R < \lfloor M(R, V)/R \rfloor + 1 \tag{6}$$

and

$$\lfloor M(R, V)/R \rfloor \leq M(R, V^*)/R < \lfloor M(R, V)/R \rfloor + 1. \tag{7}$$

Hence, $|F(p_r) - F^*(p_r)| = |M(1, V) - M(R, V^*)| < R$ for a game tree of height $h + 1$.

By induction, the theorem holds for a game tree of any height.

4 Experiments

In this section we consider the experimental environment, settings, and results. We also present an analysis of the results.

4.1 Experiments

We conducted experiments to evaluate the performance of the proposed search algorithm with and without resolution on random trees and real game trees. First, we compared the performance of different algorithms on a random tree with the leaf nodes sampled from a discrete uniform distribution. Then, we compared the performance of our algorithm with and without resolution in the Chinese chess program CONTEMPLATION with a good evaluation function [17]. Note that CONTEMPLATION uses a transposition table. The value stored in the table is its original value, not the one after applying the resolution scheme. The experiments were run on a server with an Intel Xeon X5680 3.33GHz CPU and 48 GB memory.

4.2 Experimental Settings for Random Trees

For each resolution value, we run the experiment 100 times on random trees to compute the performance of three algorithms: Negamax search with alpha-beta pruning, Negascout search, and Negascout-with-resolution. To compare the performances, we calculate the number of nodes visited by each algorithm. Using the Negamax algorithm as the baseline, we compute the number of nodes visited by Negascout and Negascout-with-resolution under different resolution levels. After running the experiment 100 times for each setting, we measure the performance by calculating the average number of nodes visited by each algorithm. In this experiment, the height of the input tree is 6; the number of branch factors is 20; and the value of each leaf node is sampled from a discrete uniform distribution between 0 and $1,000,000$.

4.3 Experimental Results and Analysis of Random Trees

The results of the experiment on the random trees are listed in Table 1. Column 1 shows the value of the resolution and column 2 indicates the number of nodes visited by the alpha-beta pruning algorithm. Columns 3 to 5 and columns 6 to 8 list the results of NegaScout and NegaScout-with-resolution respectively. Columns 3 and 6 show the number of nodes visited by the respective algorithm; columns 4 and 7 show the corresponding ratios of nodes visited by the alpha-beta

Table 1. Experiment results on random trees; for each resolution value, we compare the number of nodes visited by NegaScout and NegaScout-with-resolution with the number visited by the Alpha-beta pruning algorithm

R	Alpha-beta	NegaScout			NegaScout-with-resolution		
	Nodes	Nodes	Ratio	σ	Nodes	Ratio	σ
1	4281558	3873730	0.8980	0.1293	3873730	0.8980	0.1293
5	4321389	3883435	0.8929	0.1191	3880661	0.8921	0.1201
10	4320529	3884022	0.8933	0.1206	3872623	0.8906	0.1216
50	4336530	3969945	0.9089	0.1266	3924251	0.8991	0.1222
100	4374453	3966224	0.8996	0.1115	3878339	0.8802	0.1133
500	4390513	4003538	0.9025	0.1262	3585845	0.8111	0.1151
1000	4344442	3907857	0.8959	0.1188	3305248	0.7617	0.0928
5000	4319328	3775147	0.8672	0.1153	1772155	0.4152	0.0821
10000	4258585	3863249	0.8988	0.1136	1200303	0.2832	0.0727
50000	4335672	3940573	0.9050	0.1170	640258	0.1511	0.0287
100000	4296391	3937171	0.9080	0.1233	613013	0.1462	0.0275

pruning algorithm; and columns 5 and 8 show the respective standard deviations. In each setting, we search the whole tree to find the best possible branch.

The results show that when the value of the resolution is not more than 100, the performance of Negascout-with-resolution is almost the same as that of the original Negascout search, i.e., 90% and 88% respectively. However, when the value is over 500, the improvement is significant. Negascout-with-resolution requires 10% less time than the original Negascout algorithm when the resolution value is 500 and the absolute error divided by the range of the leaf nodes' values is only 0.0005. If we increase the resolution value to 5000, Negascout-with-resolution requires approximately 50% less time and the absolute error divided by the range of the leaf nodes' values is only 0.005.

As the leaf nodes' values are uniformly distributed between 0 and 1,000,000, increasing the resolution value also increases the chances of pruning branches, i.e., the resolution value is positively related to the number of pruning nodes. In this case, the range of the leaf nodes' values is large enough, so we can derive a good approximation within a constant error and save more than 50% of the time on the random trees.

4.4 Experimental Setting for Real Game Trees

To evaluate the performance of the resolution value on the real game trees, we begin by comparing the performance of the Aspiration search algorithm and the Aspiration search algorithm with resolution on a set of 84 benchmarks used by Contemplation. For both algorithms, null move pruning and the history heuristic are used in the program. Once again, we evaluate the performance by comparing the number of nodes visited by the two algorithms. Next, we use a self-play test in which 100 games are played against the original program without

resolution using the time limit of a Computer Olympics game, i.e., a maximum of 30 minutes for each player.

The value of the resolution in CONTEMPLATION changes dynamically according to the current search result. We use the following procedure to determine the value of the resolution. First, we consider the current depth. If it is less than 6, we set the value of the resolution at 1, and the corresponding level is the default level -1. However, if the current depth is greater than 6, we use the returned score of the previous depth, S, to determine the value of the resolution. The range of the returned value is divided into several levels; for example, if we divide the scores every 64 points, then 0 to 63 will be the first level, 64 to 127 will be the second level, and so on. There are two exceptions where we decrease or increase a level's value: (1) the difference between the returned values of the previous two searches is small enough, e.g., less than 20, so we increase the level of the resolution; and (2) the difference between the returned values of the previous two depths is higher than a threshold, e.g., half the length of one level, so we decrease the level because the current state is unstable and the search must be performed carefully. We assign a unique resolution to each score level. For example, if we divide the score every 64 points, we set the resolution of level L as $R = 2^L$, where $L = \lfloor |\text{previous score}|/64 \rfloor$.

When we use Algorithm 1 in CONTEMPLATION, the resolutions of beta cut in Line 15 and value comparison in Line 8 are set separately, i.e., there are two resolution values. Although we assign a constant error bound in the algorithm with resolution, we focus on the relation between the search depth and the resolution. In general, for a parent node of which the children are leaves, the resolution should be small. The score of each move is propagated from the leaf node to the root node, and the error is propagated in the same direction; that is, the values of the resolutions used for the child nodes affects the precision of the returned value of the parent node. Hence, as the value of the resolution used in the predecessor of a leaf node increases, the precision of the search result decreases. This means that if we want to control the error, the resolution should be small for the nodes close to the leaves.

Although we know the value of the resolution should be set according to the tree depth, we cannot determine an exact search depth for iterative deepening based on the search because it is time dependent. Since the search depth cannot be determined, we extend the idea of "nodes have a smaller resolution when they are near a leaf node" as follows. Nodes in the game tree can be separated into odd and even layers, which are moves considered by the game players, i.e., the red side and black side. If a move belongs to the red side, to find the precise score, the predecessor of the leaf node, i.e., the black side, should use the exact resolution, i.e., $R = 1$. In contrast, as the black side has the precise search results, we can assume that it does not use the resolution for all of its layers, i.e., one side searches the game tree with the resolution, but the other side does not. During an iterative deepening search, we increase the depth by two levels each time. In the algorithm, we only use the resolution when the current state is our turn, and we assume that the opposite side always searches for the precise

value without resolution. In some of the experimental settings, we assess the algorithm's performance by using the resolution in even layers only or in both layers, i.e., for one player's turns or for both players' turns. The objective is to determine whether the resolution should be used in even layers only or in both layers.

Use the Resolution Scheme Only in Beta Cut

First, we test cases where the resolution scheme is used only in beta cut. The parameter settings are listed in Table 2. Each setting is shown in two rows, which represent the value of the resolution used for the beta cut and value comparison respectively. Columns 1, 2, and 3 show the parameter settings, the resolution used, and the range of the score level respectively. The following nine columns show the value of the resolution in each level. Level −1 is used as the initial value and level 7+ contains all the levels above 7. The last column shows whether the resolution value is used in even layers only or in both layers.

Table 2. Testing different resolution for beta-cut while do not use the resolution scheme in doing value-comparison

Setting	Type	Level	-1	0	1	2	3	4	5	6	7+	odd-even
I-1	Beta-cut	100	2	2	2	2	2	2	2	2	2	even
	Value-comparison	100	1	1	1	1	1	1	1	1	1	even
I-2	Beta-cut	100	1	2	2	2	3	3	3	3	4	even
	Value-comparison	100	1	1	1	1	1	1	1	1	1	even
I-3	Beta-cut	100	1	1	1	2	2	5	5	10	20	even
	Value-comparison	100	1	1	1	1	1	1	1	1	1	even
I-4	Beta-cut	100	1	1	2	4	5	10	20	20	25	even
	Value-comparison	100	1	1	1	1	1	1	1	1	1	even

The experimental results of Group I are listed in the Table 3. Column 1 shows the parameter settings used by the program with resolution. For each benchmark and parameter setting, to obtain the ratio of nodes visited, we divide the number of nodes visited by the algorithm with resolution by the number of nodes visited by the original algorithm. Columns 2 to 4 show the number of benchmarks of which the ratios are greater than 1, less than 1, and equal to 1, respectively. Columns 5 and 6 show the ratios and the standard deviations, respectively. Columns 7 to 9 show the game results of the modified program with resolution against the original program, i.e., the number of games that the modified program wins, loses or draws, respectively. Column 10 shows the number of points each program accumulates; each win, loses or draw is worth 2 points, zero, and 1 point, respectively. Columns 11 and 12 show the winning rates and the corresponding standard deviations. For the benchmarks, the mean ratio of parameter settings in Group I is about 1.0. For parameter settings I-3, the mean ratio is less than 1. With regard to the self-play test, parameter setting I-4 yields the best result with 32 wins, 19 losses, and 49 draws.

Table 3. The experimental results of benchmark testing and self-play testing on a Chinese chess program under parameter settings Group I

Setting	Benchmark testing					Self-play testing					
	Less	More	Equal	Nodes	σ	Win	Lose	Draw	Points	Rate	σ
I-1	38	24	22	1.0091	0.3420	27	25	48	102	0.510	0.2704
I-2	12	5	67	1.0034	0.0292	23	29	48	94	0.470	0.2871
I-3	24	12	48	0.9898	0.0967	26	24	50	102	0.510	0.2460
I-4	33	28	23	1.0488	0.4552	32	19	49	113	0.565	0.2208

For the parameter settings in Group I, the results show that when the resolution is only used for beta cut, the value of the resolution increases as the player's lead in the game increases. The parameter setting with a larger resolution value for the higher score level achieves a better performance. In this case, setting I-4 yields the best performance among the first four parameter settings.

Use Same Resolution in Both Places

Next, we compare the performance where both beta cut and value comparison use the same resolution. The parameter settings are listed in Table 4 and the format is similar with Table 2.

Table 4. Using the resolution scheme on beta-cut and value-comparison

Setting	Type	Level	-1	0	1	2	3	4	5	6	7+	odd-even
II-1	Beta-cut	100	2	2	2	2	2	2	2	2	2	even
	Value-comparison	100	2	2	2	2	2	2	2	2	2	even
II-2	Beta-cut	100	1	2	2	2	3	3	3	3	4	even
	Value-comparison	100	1	2	2	2	3	3	3	3	4	even
II-3	Beta-cut	100	1	1	1	2	2	5	5	10	20	even
	Value-comparison	100	1	1	1	2	2	5	5	10	20	even
II-4	Beta-cut	100	1	1	2	4	5	10	20	20	25	even
	Value-comparison	100	1	1	2	4	5	10	20	20	25	even

The experimental results of Group II are listed in the Table 5 and the format is similar with Table 3. For the benchmarks, the mean ratio of parameter settings in Group II are slightly greater than 1.0 and parameter settings II-1 has the smallest mean ratio 1.0088. With regard to the self-play test, parameter setting II-1 yields the best result with 33 wins, 26 losses, and 41 draws.

For parameter settings in Group II, both beta cut and value comparison use the same resolution value. In this case, the result is the opposite of that under the first four parameter settings. Since the resolution is also used for value comparison, if the value of the resolution is very large, a larger number of moves would be treated as equivalent. Therefore, if the move order is poor, the

Table 5. The experimental results of benchmark testing and self-play testing on a Chinese chess program under parameter settings Group II

	Benchmark testing					Self-play testing					
Setting	Less	More	Equal	Nodes	σ	Win	Lose	Draw	Points	Rate	σ
II-1	11	16	57	1.0088	0.0573	33	26	41	107	0.535	0.3315
II-2	23	27	34	1.0091	0.1658	21	26	53	95	0.475	0.2233
II-3	34	39	11	1.1194	0.5923	15	24	61	91	0.455	0.0548
II-4	38	36	10	1.2067	0.9273	22	30	48	92	0.460	0.2902

algorithm will not perform well. Hence, the best result is achieved under parameter setting II-1, which sets all the resolutions at 2.

Use Different Resolutions

Finally, we use different resolutions for beta cut and value comparison, and determine whether all possible combinations of the resolutions should be used in even layers only or in both layers. The parameter settings are listed in Table 6 and the format is similar with Table 2.

Table 6. Using different resolutions for beta-cut and value-comparison

Setting	Type	Level	-1	0	1	2	3	4	5	6	7+	odd-even
III-1	Beta-cut	64	1	1	1	2	4	8	16	32	32	even
	Value-comparison	64	1	1	1	2	4	8	16	32	32	even
III-2	Beta-cut	64	1	1	1	2	2	4	4	8	16	even
	Value-comparison	64	2	2	2	2	2	2	2	2	2	even
III-3	Beta-cut	64	1	1	1	2	4	8	16	32	32	both
	Value-comparison	64	2	2	2	2	2	2	2	2	2	even
III-4	Beta-cut	64	1	1	1	2	4	8	16	32	32	both
	Value-comparison	64	2	2	2	2	2	2	2	2	2	both

The experimental results of Group III are listed in the Table 7 and the format is similar with Table 3. For the benchmarks, the standard deviation of parameter settings in Group III are around 0.4 and both the mean ratios of parameter settings III-3 and III-4 are less than 1. With regard to the self-play test, parameter setting III-4 yields the best result with 30 wins, 18 losses, and 52 draws.

Combining the results of Group I and Group II, we observe that, when the score is high, the resolution used for value comparison should be small, and the resolution used for beta cut should be large. Parameter settings in Group III use different values for the resolutions, and we try to determine whether the resolution value should be used in the even layers only, or in all the layers. The results of parameter setting III-4 show that our observation is correct. The resolution used for value comparison should be small, and the resolution used

Table 7. The experimental results of benchmark testing and self-play testing on a Chinese chess program under parameter settings Group III

Setting	Benchmark testing					Self-play testing					
	Less	More	Equal	Nodes	σ	Win	Lose	Draw	Points	Rate	σ
III-1	31	46	7	1.2193	0.6562	13	33	54	80	0.400	0.2303
III-2	35	44	5	1.0805	0.5823	21	33	46	88	0.440	0.3161
III-3	45	25	14	0.9109	0.2194	27	27	46	100	0.500	0.2973
III-4	51	30	3	0.9498	0.4590	30	18	52	112	0.560	0.1781

for beta cut should be large when the player has a strong advantage. The results of parameter settings III-3 and III-4 also confirm that, for identical settings, the same resolution should be used in all layers.

5 Conclusions

In this paper, we use an adaptive resolution technique to enhance min-max search with alpha-beta pruning. The modified algorithm, which has a constant error bound R on the returned value. We also observe from the experiments that the modified algorithm can find moves that are as good as the ones found by the original algorithm faster. Hence, we conjecture that the modified algorithm is robust without caring too much about small difference between scores of similar, but different boards. Because the resolution is chosen dynamically, the modified algorithm improves the performance and finds good moves quickly. Moreover, the experimental results show that this enhancement yields a significant improvement in the performance on both random trees and real game trees. The performance on random trees increases by 50% with an error bound, where the absolute error divided by the range of the leaf nodes' values is about 0.5%. The experimental results for real game trees show that the resolution of value comparison should be small and that of beta cut should be large when the score is high. Moreover, the resolution technique for beta cut and value comparison should be applied to all layers. Under the best setting, the win rate is 62.5% among all the non-drawn games using self-play.

References

1. Adelson-Velskiy, G.M., Arlazarov, V.L., Donskoy, M.V.: Some methods of controlling the tree search in chess programs, pp. 129–135. Springer-Verlag New York, Inc., New York (1988)
2. Akl, S.G., Newborn, M.M.: The principal continuation and the killer heuristic, pp. 466–473 (1977)
3. Atkin, L., Slate, D.: Chess 4.5-The Northwestern University chess program, pp. 80–103. Springer-Verlag New York, Inc., New York (1988)
4. Björnsson, Y., Marsland, T.A.: Multi-cut alpha-beta-pruning in game-tree search. Theoretical Computer Science 252(1-2), 177–196 (2001)

5. Levy., D., Broughton, D., Taylor, M.: The sex algorithm in computer chess. ICGA Journal 12(1) (1989)
6. Harris, L.R.: The heuristic search and the game of chess - a study of quiescene, sacrifices, and plan oriented play. In: Proceedings of IJCAI, pp. 334–339 (1975)
7. Heinz, E.A.: Extended futillity pruning. ICGA Journal 21(2), 75–83 (1998)
8. Jiang, A.X., Buro, M.: First experimental results of probcut applied to chess. In: Proceedings of ACG, pp. 19–32 (2003)
9. Kaindl, H., Shams, R., Horacek, H.: Minimax search algorithms with and without aspiration windows. IEEE Trans. Pattern Anal. Mach. Intell. 13, 1225–1235 (1991)
10. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artif. Intell. 6(4), 293–326 (1975)
11. Lim, Y., Lee, W.: Rankcut - a domain independent forward pruning method for games. In: Proceedings of AAAI (2006)
12. Plaat, A.: Mtd(f) a minimax algorithm faster than negascout (1997), http://people.csail.mit.edu/plaat/mtdf.html
13. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-first fixed-depth minimax algorithms. Artif. Intell. 87(1-2), 255–293 (1996)
14. Reinefeld, A.: An improvement of the scout tree search algorithm. ICCA Journal 6(4), 4–14 (1983)
15. Schaeffer, J.: The history heuristic and alpha-beta search enhancements in practice. IEEE Trans. Pattern Anal. Mach. Intell. 11(11), 1203–1212 (1989)
16. Shannon, C.E.: Programming a computer for playing chess. Philosophical Magazine 41(314), 256–275 (1950)
17. Wu, K.-C., Hsu, T.-S., Hsu, S.-C.: Contemplation wins Chinese-chess tournament. International Computer Game Association (ICGA) Journal 27(3), 172–173 (2004)

Designing Casanova: A Language for Games

Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Giulia Costantini,
Michele Bugliesi, and Mohamed Abbadi

Università Ca' Foscari Venezia
DAIS - Computer Science
{maggiore,spano,orsini,costantini,bugliesi,mabbadi}@dais.unive.it

Abstract. Games are complex pieces of software which give life to animated virtual worlds. Game developers carefully search the difficult balance between quality and efficiency in their games.

In this paper we present the Casanova language. This language allows the building of games with three important advantages when compared to traditional approaches: simplicity, safety, and performance. We will show how to rewrite an official sample of the XNA framework, resulting in a smaller source and a higher performance.

1 Introduction

Computer games promise to be the next frontier in entertainment, with game sales being comparable to movie and music sales in 2010 [5]. The unprecedented market prospects and potential for computer-game diffusion among end-users have created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. Our present endeavor makes a step along these directions.

Making games is a complex business. Games are large pieces of software with many heterogeneous requirements, the two crucial being high quality and high performance [2]. High-quality in games is comprised by two main factors: visual quality and simulation quality. Visual quality in games has made huge leaps forward, and many researchers continuously push the boundaries of real-time rendering towards photorealism. In contrast, simulation quality is often lacking in modern games; game entities often react to the player with little intelligence, input controllers are used in straightforward ways and the logic of game levels is more often than not completely linear. Building a high-quality simulation is rather complex in terms of development effort and also results in computationally expensive code. To make matters worse, gameplay and many other aspects of the game are modified (and often even rebuilt from scratch) many times during the course of the development. For this reason game architectures require a large amount of flexibility.

To manage the complexity, game developers use a variety of strategies. They have used object-oriented architectures, component-based systems, and reactive programming, with some degree of success for this purpose [6,7,4].

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 320–332, 2012.
© Springer-Verlag Berlin Heidelberg 2012

In this paper we present the Casanova language, a language for making games, as a solution to the obstacles mentioned above. Casanova offers a mixed declarative/procedural style of programming which has been designed in order to facilitate game development. The basic idea of the language is to require from the developer only and exclusively those aspects of the game code which are specific to the game being developed. The language aims for simplicity and expressive power, and thanks to automated optimizations it is capable of generating code that is much faster than hand-written code and at no effort for the developer. The language offers primitives to cover the development of the game logic, and incorporates the typical processing of a game engine. Also, the language is built around a theoretical model of games with a "well-formedness" definition, in order to ensure that game code is always a good model of the simulated virtual world.

In the remainder of the paper we show the Casanova language in action. We begin with a description of the current state of game engines and game programming in Section 2. In Section 3 we define our model of games. We describe the Casanova language in Section 4. We show an example of Casanova in action, and also how we have rewritten the game logic of an official XNA sample from Microsoft [18] in Casanova with far less code and higher runtime performance in Section 5. In Section 6 we discuss our results and some future work.

2 Background

In this section we discuss five current approaches to game development. The two most common game engine architectures found in today's commercial games are (1) object-oriented hierarchies and (2) component-based systems. In a traditional object-oriented game engine the hierarchy represents the various game objects, all derived from the general `Entity` class. Each entity is responsible for updating itself at each tick of the game engine [1]. A component-based system defines each game entity as a composition of components that provide reusable, specific functionality such as animation, movement, and reaction to physics. Component-based systems are being widely adopted, and they are described in [6].

These two, approaches are rather traditional and suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations in a dynamic, even composable way. By doing so, they lose the focus on the fact that most entities in a game need to *interact* with one another (collision detection, AI, etc.). Usually much of a game complexity comes from defining (and optimizing) these interactions. Moreover, all games feature behaviors that take longer than a single tick; these behaviors are hard to express inside the various entities, which often end up storing explicit program counters to resume the current behavior at each tick.

There are two more approaches that have emerged in the last few years as possible alternatives to object-orientation and component-based systems. They are (3) (functional) reactive programming and (4) SQL-style declarative programming.

Functional reactive programming (FRP, see [4]) is often studied in the context of functional languages. FRP programming is a data-flow approach where value modification is automatically propagated along a dependency graph that represents the computation. FRP offers a solution to the problem of representing long-running behaviors, even though it does not address the problem of many entities that interact with each other.

SQL-queries for games have been used with a certain success in the SGL language (see [15]). This approach uses a lightweight, statically compiled query engine for defining a game. This query engine is aggressively optimized, in order to make it to express efficient aggregations and cartesian products, two very common operators in games. In contrast, SGL suffers when it comes to representing long-running behaviors, since it focuses exclusively on defining the tick function.

With these two issues in mind we have designed Casanova. Casanova is the fifth approach of this section. It seamlessly supports the integration of the interactions between entities and long-running behaviors.

3 A Model for Games

We define a game as a triplet of (1) a game state, (2) an update function, and (3) a series of asynchronous behaviors. In this model we purposefully ignore the drawing function, since it is not part of the current design of Casanova.

```
type Game 's =
  { State : 's; Update : 's -> DeltaTime -> (); Behavior : 's -> () }
```

The game state is a set of (homogeneous) collections of entities; each entity is a collection of attributes, which can either be *(i)* primitive values, *(ii)* collections of attributes or *(iii)* references to other entities.

The update function modifies the attributes of the entire state according to a fixed scheme which does not vary with time; we call this fixed scheme the **rules** of the game; rules can be physics, timers, scores, etc. Each attribute of each entity is associated to exactly one rule. The update function is quick and must terminate after a brief computation, since it is invoked in a rather tight loop that should perform 60 iterations per second.

The behavior function is a sequential process which performs a series of operations on the attributes of the game entities. It is a long-running, asynchronous process with its own local state, it runs in parallel with the main loop and it can access the current clock time at any step to perform actions which are synchronized with real time. The processing over the game state takes more than one tick; behaviors are used, for example, for implementing AIs.

A game engine is thus a certain way of processing a game (see the box below).

```
let run_game (game:Game 's) =
  let rec run_rules (t:Time) =
    let t' = get_time()
    game.Update game.State (t'-t)
    run_rules t'
  parallel (run_rules (get_time()), game.Behavior(game.State))
```

We define four properties of a correct and well-behaving game: *(i)* each entity is updated exactly once per tick, *(ii)* the entity update is order-independent, *(iii)* the tick always terminates, and *(iv)* the game runs at an interactive frame rate.

Casanova guarantees only the first three requirements. The fourth requirement cannot be guaranteed, since it heavily depends on factors, such as the size of the virtual world and the computational resources of the machine used to run the game; nevertheless, by automating certain optimizations Casanova makes it easier to achieve the fourth requirement. Below we discuss the architecture of a Casanova game.

Architecture of a Casanova Game
Behaviors are used to make it easier to handle complex input and to build articulated level logics or customized AI algorithms into the game. While Casanova does not (yet) integrate any deduction engine or proper AI system, it makes integrating such a system with the game loop and the game state much simpler.

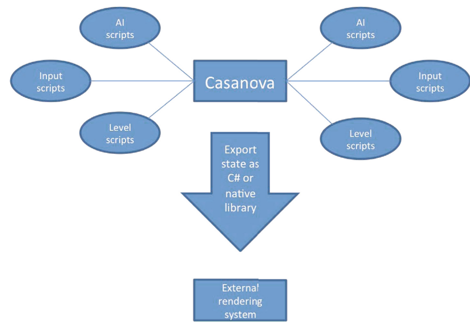


Fig. 1. Game architecture with Casanova

Rules are used to build all the regular logic that the game continuously repeats. An example is the fact that when projectiles collide with an asteroid then the asteroid is damaged or other logical relationships between entities occur. Rules are the main workhorse of a game, and Casanova ensures that all the queries that make up the various rules maintain the integrity of the state and are automatically optimized to yield a faster runtime.

The Casanova compiler will export the game state as a series of type definitions and classes that can be accessed directly (that is without any overhead) from a C# or C++ rendering library; in this way it takes little effort to integrate the existing rendering code and engines with the help of Casanova.

4 The Casanova Language

In this section we present the Casanova language; for a more detailed treatment, we refer to [11]. Casanova is inspired to the ML family of languages. We first discuss the design goals (4.1), then provide a brief introduction (4.2), followed by a description of syntax, semantics, and types (4.3). In 4.4 an introductory example is given. Optimization is described in 4.5 and a full example is given in 4.6.

4.1 Design Goals

We have designed the Casanova language with multiple goals in mind. First of all, Casanova games must be easy and intuitive to describe. For this reason we have used a mix of declarative and procedural programming. For expressing rules, declarative programming is cristal clear, allows the developer to focus on what he wants to achieve rather than how, and there is a wealth of powerful optimization techniques for declarative operations on sequences of values coming from the field of databases [8]. The declarative portions of a game are all executed in parallel, and can take advantage of multi-core CPUs.

Procedural programming, in particular coroutines [9], are used to describe computations that take place during many ticks of the game engine. Imperative coroutines are used to express the behaviors of a game. These behaviors are executed sequentially and with no optimizations, since they can access any portion of the state both for reading and writing, and they may perform any kind of operation.

4.2 A Brief Introduction to Casanova

Casanova is a programming language designed around a set of core principles aimed at aiding game development. Here we describe the language "at a glance", by listing its features designed to simplify repetitive, complex or error prone game coding activities: *(i)* Casanova integrates the game loop and time as first-class constructs. The game loop and time management are almost always an important part of game development libraries, for example see [17]; *(ii)* it performs a series of optimizations that are usually found hand-coded in virtually all game engines [2], such as logical optimization of queries on lists and spatial partitioning/use of indices to speed up quadratic queries, such as collision detection (for example: `colliders(self) = [other | other <- Others, collides(self,other)]`; *(iii)* it guarantees that updates to the game state during one tick are consistent, that is, the state is never partially updated thanks to a (high-performance) transactional system; *(iv)* it offers a scripting system that integrates seamlessly with the update loop.

We have designed Casanova with the aim of adding more features such as: *(i)* automated generation of all the rendering code; *(ii)* automated generation of all the networking code; *(iii)* automated generation of all or parts of an AI system.

Of course, the language can also serve as a general purpose language. Any application that requires performing computations and visualization on a complex set of data which evolves over time according to a set of fixed rules might benefit from using Casanova. In the future, we may investigate other possible uses of the language in this direction. As a final remark, it must be noted that Casanova sometimes constrains the developer; for example, at most one rule may be associated with any given field of the game state and rules are always applied at every tick of the simulation. Since developers may find this set of restrictions too tight we have included a scripting system which can also act as a "wildcard" in this regard, that is scripts have essentially no limitations in expressivity

(scripts are a general purpose programming language with coroutines) and for this reason they can be used to express anything that the rule system cannot, albeit renouncing various useful features such as automated optimization.

4.3 Syntax, Semantics, and Types

The details of the Casanova language syntax, semantics, and type system are defined in [11]. In this subsection we give a general overview of the most salient aspects of the language.

A Casanova program is divided into three parts: *(i)* the state definition, *(ii)* the initial state, and *(iii)* the main behavior.

The state definition contains the type definitions of the game state and game entities, together with the rules to which the various fields are subjected. Rules may be nested, i.e., a field may contain a rule of type `Rule T`, where `T` contains a value of type `Rule V`. This is quite common, and we will seen an instance of this in the example in subsection 4.4.

Entities and the state may be defined in terms of the usual type constructors found in a functional language: records, tuples, and discriminated unions. Also, we can define values of type: table (for sequences), variable (for mutable cells), rule (for updateable fields), and reference (for read-only pointers).

The initial state defines the starting value of the various game entities. The main behavior is an imperative process which runs for the entire duration of the game. A behavior may spawn (`run`) other behaviors, suspend itself for one or more ticks (`yield` or `wait`) or wait for another behavior to complete before resuming its execution (`do!` or `let!`). In addition, behaviors may access the state without any limitation; a behavior can read or write any portion of the state: `:=` is the assignment operator and `!` is the lookup operator.

Behaviors can be combined with a small set of operators that define a straightforward concurrent calculus: `parallel x y`, which runs two behaviors in parallel and returns the pair with their results; `concurrent x y`, which runs two behaviors in parallel and returns the result of the first to terminate; `x => y`, which runs behavior `y v` only when `x` terminates with result `Some v`; and `repeat x`, which continuously runs a behavior.

The tick function of the game is built automatically by the Casanova compiler, and it executes all running behaviors until they `yield`; then it executes all rules (in parallel and without modifying the current game state to avoid interferences); finally it creates the new state from the result of the rules.

Rules do not interfere with each other, since they may not execute imperative code. If rules immediately modified the current state, then their correctness would depend on a specific order of execution. Specifying said order would place an additional burden on the programmer's shoulders.

The tick function for rules presents a problem which is partly addressed with `references`: portions of the state must not be duplicated, for correctness reasons. This means that each entity in Casanova may be subjected to some rules but only once; if an entity is referenced more than once then it may be subjected to more (and possibly even contradictory) rules. For this reason we make any

value of type Rule (or which contains a field of type Rule) linear [13]. This means that a value of type Rule T may be used at most once, and after it is read or used it goes out of scope.

We use the type constructor Ref T to denote a reference to a value of type T. A reference is a shallow copy to an entity which primary value is stored elsewhere. This allows for the explicit sharing of portions of the game state without duplication of rules, since rules are not applied to references. This also allows for safe cyclical references, such as given below.

```
type Asteroid = { ... Colliders : Rule(Table(Ref Projectile)) }
type Projectile = { ... Colliders : Rule(Table(Ref Asteroid)) }
```

This restriction is enforced statically during type checking, and it ensures that all rules are executed exactly once for each entity. The type checker enforces another property: a behavior gives a compile-time error unless it is statically known that all code paths yield. This is achieved by requiring that repeat and => are never invoked on a behavior which does not yield in all its paths. An example is the behavior below.

```
repeat { if !x > 0 then yield else y := 10 }
```

which will generate a compile-time error.

This ensures that the tick function will always terminate, because rules are non-recursive functions and behaviors are required never to run without yielding indefinitely.

Of course, it is possible to lift this restriction, since it may give some false negatives; for this reason, the actual Casanova compiler will be configurable to give just a warning instead of an error when it appears that a script does not yield correctly, to leave more freedom to those developers who need it.

So far the Casanova language enforces the following four properties.

– developers do not have to write the boilerplate code of traversing the state and updating its portions; this happens thanks to the fact that Casanova automatically builds the game loop
– all entities of the state are updated exactly once (even though they may be shared freely across the state as Refs); this happens thanks to the linearity of the Rule data type and the automatic execution of all rules by the game loop
– rules do not interfere and are processed simultaneously; this happens thanks to the linearity of the Rule data type and thanks to the fact that the state is created anew at each tick
– the tick function always terminates; this happens because the state is not recursive (again, thanks to the linearity of Rule) and because our coroutines are statically required always to invoke yield

These properties alone are the correctness properties and ensure that the game will behave correctly. We will now see an example Casanova game. We will also see the set of optimizations implemented by the Casanova compiler. They make sure that a game runs fast with no effort on the part of the developer.

4.4 Introductory Example

A Casanova program starts with the definition of the game state, the various entities and their rules. A field of an entity may have type `Rule T` for some type `T`. This means that such a field will contain a value of type `T`, and will be associated with a function of type: $\texttt{Ref(GameState)} \times \texttt{Ref(Entity)} \times \texttt{T} \times \Delta\texttt{Time} \rightarrow \texttt{T}$

This function is the *rule function*, and its parameters are (they can be omitted by writing an underscore _ in their position): *(i)* the current state of the game; *(ii)* the current value of the entity we are processing; *(iii)* the current value of the field we are processing; *(iv)* the number of seconds since the last tick.

When a field does not have an explicit rule function, then the identity rule is assumed. A rule function returns the new value of a field, and cannot write any portion of the state. Indeed, the current value of the state and the current entity are read-only inside the body of a rule function to avoid read-write dependencies between rules. Updating the state means that all its rule functions are executed, and their results stored in separate locations. When all rule functions are executed, then the new state is assembled from their results.

In the remainder of the paper we will omit some type annotations; this is possible because we assume the presence of type inference. In a field declaration, the `:` operator means "has type", while the `::` operator specifies the rule function associated with a rule. The `!` operator is the dereferencing operator for rules, and it has type `Rule T -> T`.

Below we show how we would build a staightforward game where asteroids fall down from the screen and are removed when they reach the bottom of the screen.

```
type Asteroid = {
    Y     : Rule float :: fun (_,self,y,dt) -> y + dt * self.VelY
    VelY  : float
    X     : float }

type GameState = {
    Asteroids
        : Rule(Table Asteroid)
        :: fun (_,_,asteroids,_) -> [a | a <- asteroids && a.Y > 0]
    DestroyedAsteroids
        : Rule int
        :: fun (_,self,destroyed_asteroids,_) -> destroyed_asteroids +
             count([a | a <- !self.Asteroids && a.Y <= 0]) }
```

In the state definition above we can see that the state is comprised by a set of asteroids which are removed when they reach the bottom. Removing these asteroids increments a counter, which is essentially the "score" of our pseudo-game. Each asteroid moves according to its velocity.

The initial state is then provided as follows.

```
let state0 = { Asteroids = []; DestroyedAsteroids = 0 }
```

Behaviors in Casanova are based on coroutines, that is they are imperative procedures which may invoke the `yield` operator. Yielding inside a behavior suspends it until the next tick of the game. Behaviors may freely access the

state for writing, that is behaviors are less constrained than rules but for this reason they also support less optimizations. The only requirement that Casanova enforces in behaviors is that they must never iterate indefinitely without yielding, and this requirement is verified with a dataflow analysis.

When the main behavior of a game terminates, the game quits. The main behavior of our game spawns asteroids every 1-3 seconds until the number of destroyed asteroids reaches 100. The main behavior of our game is defined as follows.

```
let main state =
  let rec behavior() = {
      do! wait (random.Next(1,3))
      state.Asteroids.Add { X = random(-1,+1); Y = 1; VelY  = random
           (-0.1,-0.2) }
      if !state.DestroyedAsteroids < 100 then do! behavior() else return ()
            }
  in behavior()
```

The imperative syntax loosely follows the monadic [12,14] syntax of the F# language, where a monadic block is declared within `{}` parentheses, and combining behaviors is done with either `do!` or `let!` and returning a result is done with the `return` statement. This allows us to mark clearly the points where a behavior waits for another behavior to complete before taking its result and proceeding.

4.5 Optimization

Casanova is designed to facilitate the automatic execution of three main optimizations: memory recycling, rule parallelization, and query optimization.

Memory recycling is a straightforward yet effective optimization for all those platforms (such as the Xbox 360) with a slow garbage collector [16]. Memory recycling means that `Rule T` fields allocate a double buffer for storing both the current and the next value for rules, instead of regenerating a new state at each tick. Rule parallelization is made possible by the static constraint that rules are linear: this means that no rules write the same memory location. We also know that rules may not freely write any references. These two facts guarantee thread safety, i.e., we may run all rules in parallel. The final optimization is query optimization. Nested list comprehensions (also known as "joins" in the field of databases [8]) can have high computational costs, such as $O(n^2)$, for example when finding all the projectiles that collide with asteroids. Such a complexity is unacceptable when we start having a large number of asteroids and projectiles, because it may severely limit the maximum number of entities supported by the game. We use the same physical optimization techniques used in modern databases: we build a spatial partitioning index (such as a quad-, oc-, R-tree) to speed up our collision detection. The resulting complexity of the same query with a spatial partitioning index is $O(n \log n)$, which executes much faster and allows us to support larger numbers of entities.

4.6 A Full Example

Below we show a full example of a game where a series of balls are thrown from one side of the screen and bounce towards the other side; the balls are removed when they reach the other side of the screen.

We start by defining the state (a collection of balls) and its rules (gravity, motion, and removal of those balls that reach one side of the screen).

```
let g = Vector2(-9.81,0.0)

type BallsState = {
    Balls       : Rule(Table Ball))
                :: fun (_,_,balls,_) -> [b | b <- balls && b.X <= 50.0 ] }
type Ball = {
    Position    : Rule Vector2
                :: fun (_,ball,p,dt) ->
                       if p.Y < 0.0 then Vector2(p.X, 0.0)
                       else p + !ball.Velocity * dt

    Velocity    : Rule Vector2
                :: fun (_,ball,v,dt) ->
                       if p.Y < 0.0 then Vector2(v.X, -v.Y) * 0.6
                       else v + g * dt }
```

Then we define the initial state, which does not contain any balls.

```
let state0 = { Balls = [] }
```

Finally we define the main behavior which launches the balls, one every second.

```
let rec main state = {
    do! wait 1.0
    state.Balls.Add { Position = Vector2(0.0, 0.0); Velocity = Vector2(5.0,
           10.0) }
    do! main state }
```

5 Case Study

In this section we will describe (1) how we have rewritten the XNA Spacewar [18] sample in Casanova, (2) the resulting reduction in code, and (3) the increases in performance obtained. We have chosen Spacewar because it is small enough to be didactically useful while being built as a starter kit, that is a starting point to be edited and extended into a different game and not just as a sample or tutorial; from this point of view, Spacewar should be considered as a small, yet complete and well-built, game.

The Casanova compiler is still in its early stages, and as such it is not yet ready for the task. The definition of the compiler can be followed by hand, and since the first Casanova compiler will generate F# code, we have written such code by hand as the compiler would have output it.

5.1 Rewriting the Game

The original sample features two ships that shoot each other while dodging asteroids that float around the gaming area. A star in the center of the playing

field pulls the players with its gravity. The first player to destroy the other (by hitting him or by making him crash on another celestial body) wins the stage.

The game state is defined as the two players, their ships, the table of asteroids and projectiles, and the sun. Also, the state contains the current gameplay status, which can either be `Playing` or `GameOver w` where `w` is the winner.

The source code of the original sample plus our implementation can be found in [3]; the current implementation of the Casanova compiler is incomplete, and at the time of writing the type checker and the F# code generator are both producing their first correct outputs but are not yet integrated together. The details of the porting are discussed in detail in [11], and we omit them here for reasons of space.

We have slightly modified the original sample so that testing could be automated. For this reason we have removed the 30 seconds time limit of each level, we have removed the victory and ending conditions, we have automated ships movement and shooting and we have increased the maximum number of asteroids and projectiles to 12 and 200 respectively. This way we have obtained an automated stress test.

We have also removed all rendering features, to avoid benchmarking rendering algorithms: Casanova does not generate rendering code, so such a comparison would have been meaningless; also, Casanova can be integrated with the very same C# rendering code of the original Spacewar. We compare the resulting frame rates to see how many simulation steps per second the original game logic is capable of performing versus the number of steps per second of the Casanova game logic; the higher this number, the more efficient the game logic and the more time remains for each frame to perform complex rendering.

As a final remark, it is worth noticing that while the original sample includes more than one thousand lines of code the length of the corresponding Casanova program is 348 lines long. The Casanova source easily fits a few pages, while navigating the original source may prove a bit complex because of its sheer size.

5.2 Resulting Benchmarks

We have benchmarked the modified sample on both the Xbox 360 and a 1.86 Ghz Intel Core 2 Duo with an nVidia GeForce 320M GPU and 4GB of RAM. In the table below we can see the frame rates of the various tests.

Table 1. Framerate of the original Spacewar vs the Casanova implementation

C# XBox	C# PC	Casanova XBox	Casanova PC
8	9	22	577

As we can see, full Casanova optimization always beats the original source by at least a factor of 2. The Xbox implementation suffers from the generation of garbage, which is a known problem of the XNA implementation on the console [16]; indeed, profiling the garbage collector shows that large amounts of

temporary memory are being generated by the program. It is noticeable that on the PC, thanks to the full optimizations done by Casanova, performance increased by almost two orders of magnitude: such an impressive increase was quite unexpected even by us, it is the more so when keeping in mind that those optimizations will be automated by the compiler.

6 Conclusions

In this paper we have presented the design of the Casanova language, a hybrid declarative/procedural language for making games. The language has a triple focus on (1) simplicity, (2) correctness (to increase developer productivity, given the complexity of game development), and (3) performance (to ensure high framerates).

We have defined a model that generalizes an abstract game, and we have introduced four important properties that describe a good game. We have shown how the Casanova language respects these properties, that is:

– rules are applied exactly once for each entity,
– rules are order-independent,
– ticks always terminate,
– automated optimizations ensure fast execution.

Our first goal is to implement a fully working prototype of the Casanova compiler that outputs F# code. The compiler is still in its early stages, and much work is still needed to achieve this goal.

Further (and less obvious) improvements may be adding support for rendering, networking, and (fully or partially) automated AI. A second venue that we are investigating is the support for reusable libraries of ready-made components, possibly with some form of statically resolved polymorphism (maybe similar to Haskell type classes) for performance reasons. Integration with an existing IDE (such as MonoDevelop or Visual Studio) is an important addition to a modern language. Finally, addressing the problem of generating less garbage (especially for the XBox 360 and for other platforms such as Windows Phone 7 and iOS through Mono) is another of our objectives.

As a final remark, some aspects of Casanova (namely scripting) already have been fully implemented, as described in [10].

References

1. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. Inf. Softw. Technol. 49, 445–454 (2007), http://dl.acm.org/citation.cfm?id=1230152.1230366
2. Buckland, M.: Programming Game AI by Example. 1 edn. Jones & Bartlett Publishers (September 2004), http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/1556220782
3. Casanova: Casanova project page, casanova.codeplex.com/

4. Elliott, C., Hudak, P.: Functional reactive animation. SIGPLAN Not. 32, 263–273 (1997), http://doi.acm.org/10.1145/258949.258973
5. ESA: Entertainment software association, http://www.theesa.com
6. Folmer, E.: Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines? In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 66–73. Springer, Heidelberg (2007), http://portal.acm.org/citation.cfm?id=1770657.1770663
7. Gameobjects: Inheritance vs aggregation in game objects, http://gamearchitect.net/Articles/GameObjects1.html
8. Garcia-molina, H., Ullman, J.D., Widom, J.: Database System Implementation (2000)
9. Knuth, D.E.: The art of computer programming: Fundamental Algorithms, 3rd edn., vol. 1. Addison Wesley Longman Publishing Co., Inc., Redwood City (1997)
10. Maggiore, G., Costantini, G.: Friendly F# (fun with game programming). Smashwords (2011)
11. Maggiore, G., Orsini, R., Bugliesi, M.: Casanova: a declarative language for safe games. Tech. Rep. 2011-7, Ca' Foscari - DAIS (2011)
12. Moggi, E.: Notions of computation and monads. Information and Computation 93, 55–92 (1989)
13. Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods. North (1990)
14. Wadler, P.: Comprehending monads. In: Mathematical Structures in Computer Science, pp. 61–78 (1992)
15. White, W., Demers, A., Koch, C., Gehrke, J., Rajagopalan, R.: Scaling games to epic proportions. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD 2007, pp. 31–42. ACM, New York (2007), http://doi.acm.org/10.1145/1247480.1247486
16. XBOX: Slow garbage collection on the xbox, http://blogs.msdn.com/b/shawnhar/archive/2007/06/29/how-to-tell-if-your-xbox-garbage-collection-is-too-slow.aspx
17. XNA: The xna framework, http://msdn.microsoft.com/xna
18. XNA: Xna spacewar 4, http://create.msdn.com/education/catalog/sample/spacewar

Affective Game Dialogues

Using Affect as an Explicit Input Method in Game Dialogue Systems

Michael Lankes[1] and Thomas Mirlacher[2]

[1] University of Applied Sciences Upper Austria, Austria
michael.lankes@fh-hagenberg.at
[2] IRIT, Toulouse, France
thomas.mirlacher@irit.fr

Abstract. Natural game input devices, such as Microsoft's Kinect or Sony's Playstation Move, have become increasingly popular and allow a direct mapping of player performance in regard to actions in the game world. Games have been developed that enable players to interact with their avatars and other game objects via gestures and/or voice input. However, current technologies and systems do not tap in the full potential of affective approaches. Affect in games can be harnessed as a supportive and easy to use input method.

This paper proposes a design approach that utilizes facial expressions as an explicit input method in game dialogues. Our concept allows players to interact with Non Player Characters (NPCs) by portraying specific basic emotions. Similar to adventure games, the player may choose between different dialogue options, which are displayed in textual form. The possible answers are coded in a way so that they can be selected by distinct facial expressions. The player may, for example, choose to act aggressively towards an NPC by expressing anger. In contrast to traditional techniques, in game dialogue systems, where players solely make their decisions by selecting text information, the proposed approach includes an affective component to reduce misunderstanding of the provided information.

A comparative study was conducted that included our interaction design as well as a traditional approach (selection of options via mouse) in order to identify possible differences and benefits in regard to the User Experience (UX). Results indicate that the use of explicit facial expressions in the context of game dialogue appears to be quite promising.

1 Introduction

According to Nacke[10], new gaming consoles and especially their novel interaction approaches require new methodologies for investigating the interaction process in video games. Although computer game technologies (input devices, game engines, asset creation tools, etc.) took a huge leap forward regarding interactivity, usability, mobility, and performance of systems, considerations dealing

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 333–341, 2012.
© Springer-Verlag Berlin Heidelberg 2012

with affective and/or social aspects played and still play only a minor role in the game design and production process [7].

The incorporation of affective and/or social aspects would offer some benefits: users are able to utilize familiar communication mechanisms when interacting with computational systems (e.g., games). Hence, the human-machine interaction process should be designed to resemble human interpersonal interaction in order to rely on skills obtained from human-human communication. Systems become easier to use if the interaction between human and machine is similar to human-to-human interaction [2]. The increase of complexity concerning games on various levels (art, design, technology, but also from a narrative point of view) requires novel forms of interaction. Elaborated game characters that should represent believable personalities and convincingly portray emotions are aimed to be part of the game world. According to Hudlicka[7], the way to achieve a rich gaming experience while interacting with game characters is to focus on affective aspects ranging from recognition, expression to the effects of emotion.

2 Affect in Game Dialogue Systems

Using facial expressions in games as an input method enables several design opportunities. Lankes et al.[9] make use of explicit facial expression in a game setting. They propose a game design approach that utilizes facial expressions as an input method under different emotional feedback configurations. A study was conducted to assess the game "EmoFlowers", focusing on User Experience (UX) and user effectiveness. The game "EmoFlowers" features facial expressions as an input method to influence the growth of a virtual flower. By displaying specific emotions, a player can alter the current weather conditions in the game. The goal of the game consists of achieving the appropriate weather condition to provide an optimal growth environment for the flower within a predefined time frame (3 minutes). The study revealed that 92,4% of the participants reported to have exprienced a positive UX while playing. 63,8% of players described that the game had an impact on their emotional disposition. The overall design of the interface was also perceived positively by 92,4% of the participants.

The integration of explicit emotional input offers also research possibilities in the field of dialog systems within games. Dialogues can be defined as a mean for participants to exchange information, achieve mutual understanding, and set up social relationships [1]. Brusk and Bjork [3] differentiate between several dialogue types in games. They distinguish between game (G), player (P), player character (PC) and non-player character (NPC): P-G (game may be controlled via gestures or voice), P-PC (players control their avatars with dialogue), NPC-NPC (comments on game status), and P-NPC (social interaction between actors). Our contribution addresses dialogues between players and NPCs (P-NPC) as it represents a typical scenario where facial expressions are performed. Facial movements and expressions appear in various forms during conversations [11]. They may emerge as punctuators (grouping sequences of words into discrete phrases), manipulators (biological needs of the face) or as regulators (control

the flow of speech). Our work focuses on the role of facial expressions in regard to conversational signals, which clarify and support what is being said.

Several attempts have been made to enhance dialogue systems in regard to the presentation and type of interaction. A majority of these approaches in games utilizes a limited number of textual answers displayed when interacting with Non Player Characters (NPCs). Current interface designs range from top down lists to more recent design such as ring interfaces. Since an adequate font size for readability is required and the screen real-estate is limited, games like Mass Effect (see Figure 1), display shorter textual answers in comparison to the actual spoken verbal information. The recently released LA Noire, a game that explicitly incorporates the interpretation of NPCs facial expressions as a core mechanic, is made up of simplified answer possibilities. The used dialogue system consists of three categories (truth, doubt, lie) that do not reflect the actual dialogue that is spoken by the player character. A second example is the Nintendo DS game Golden Sun: Dark Dawn that offers players to interact solely with NPCs via four emoticons. The problem that occurs in this dialogue system is that players do not know to which aspect an emoticon is referring to. The lack of detail, especially the absence of emotional aspects, can lead to misunderstandings, or in the worst case, to no understanding at all.

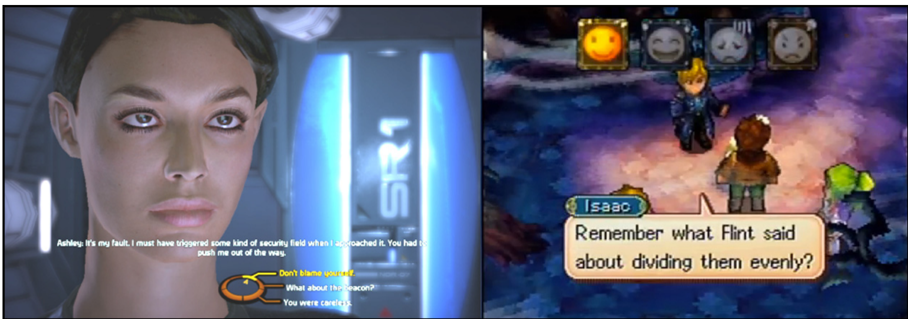


Fig. 1. Examples of interfaces in the context of game dialogue - left: Mass Effect, right: Golden Sun: Dark Dawn

In contrast, researchers such as Zhan et al. [12] propagate complex systems that incorporate a real-time system for recognizing player's facial expressions. These systems can be used to control avatars' emotional states by directly controlling the animation engine instead of issuing text commands. Cavazza et al. [4] propose an approach to user interaction with virtual characters entirely based on emotional speech. The authors claim that affective interaction allows unconstrained linguistic expression, as part of a dialogue with the feature character of the interactive narrative. However, the researchers remark that the dialogue is limited to pairs of utterances, without any extended dialogue phenomena. Limitations of this approach arise from the fact that their impact depends on genre considerations. These approaches appear to be fairly complex (both for developers and users).

We propose a solution that incorporates a straightforward way of interaction, while, at the same time, it offers the complexity and potential of affective solutions. Our approach is based on a single initiative, turn-taking system [3]. Turn taking systems form basic dialogue control elements that address how people define their actions in a conversation. The term single-initiative dialogues means that a dialogue can only be started and/or moderated by one participant. Our concept introduces a design that utilizes facial expressions as an explicit input method (see Figure 2). These concept allows a player to interact with NPCs by using specific basic emotions (joy and anger). These emotions expressed via facial expressions serve as an selection tool to choose amongst several available answers. Similar to the previously described concepts, the player may choose between different dialogue options, which are displayed as color-coded texts. Answers with a friendly attitude towards the NPC are displayed with a yellow color and a smiley emoticon in the center of a ring interface. If the player opts to use a more aggressive strategy, then a red text and the emoticon with the anger expression are chosen. Players have to press and hold the capture button (space key) to activate the detection of the facial expression. This was done in order to avoid unwanted facial expression input. To confirm a facial expression, a player needs to perform the expression for 2 seconds. The process is supported by the interface via a circle in the center of the ring interface. When the capture mode is off the emoticon has 50% transparency and grey color and does not depict any

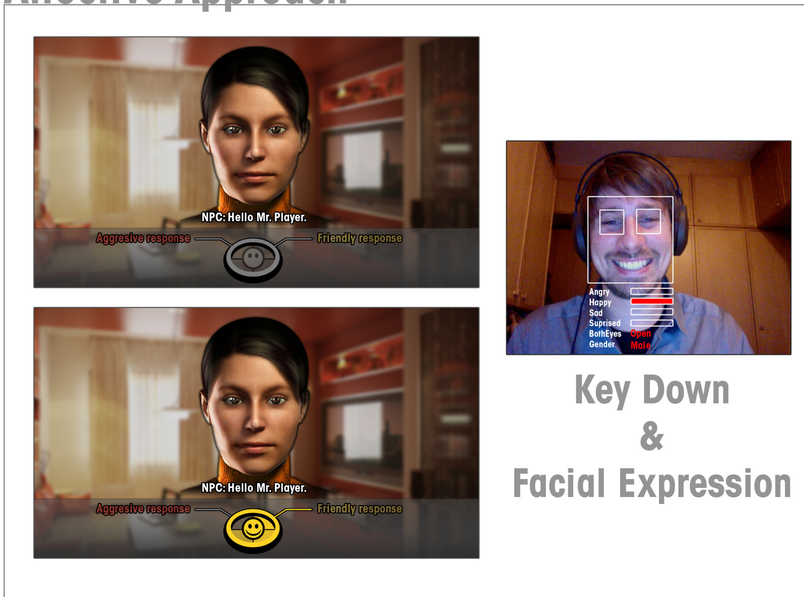


Fig. 2. Affective approach: by pressing and holding the SPACE key and by performing a facial expression (joy or anger) the player chooses a dialogue option

expression. When either joy or anger is detected, the emoticon changes its type (joy or anger) and is set to 100% opacity. Furthermore, the circle in the center is filled with either red or yellow color. If the circle is completely filled, an auditive cue is played and the next dialogue step will be presented.

The technical setup incorporates all components into a MacBook Pro: the EmoTracker and the Graphical User Interface (GUI) module. The input part of the EmoTracker module captures raw images from a built-in webcam and sends these images to the Facedetector [5]. This Facedetector in turn finds faces in the input image, and evaluates basic emotions (joy, anger) within each detected facial expression [8]. After detecting faces and facial expressions, the first face found along with the detected properties (emotions) is passed on as XML data transmitted via a loopback Transmission Control Protocol (TCP) stream to the GUI module. The GUI module is an adobe flash application, which handles the display of the user interface as well as the selection within the dialog system.

3 Evaluation

Our design goal was to create a system that features an easy to learn way of interaction as well as the integration of affective input. We aimed for a positive UX, especially we wanted to address hedonic aspects, as task related (pragmatic) aspects are not the only important component when playing games. To perform an evaluation of our concept accordingly (and to provide a comparison) we set up a second system that resembles a traditional approach (see Figure 3). In this approach players interact with NPCs via mouse input. Analogous to the affective approach, answers are color coded (red: anger, joy: yellow), but, in contrast to our design, they are selected with mouse clicks. This type of interaction can be also seen in games such as Mass Effect or Dragon Age: Origins.

3.1 Measurement Tool

As we wanted to have a straightforward and flexible tool to evaluate the impact of our concepts, we decided to measure the UX with the AttrakDiff questionnaire [6]. It has been effectively used in various studies to investigate the perceived pragmatic and hedonic quality of interactive systems. The AttrakDiff questionnaire was developed to measure implications of attractiveness of a product. Users indicate their impression of a given product by bipolar terms, categorized into four dimensions. The first dimension, the Pragmatic Quality (PQ), describes traditional usability aspects, while the dimension Hedonic Quality-Stimulation (HQ-S) refers to the need of people for further development concerning themselves. By supporting this aspect, products can offer new insights and interesting experiences. Hedonic Quality-Identification (HQ-I) allows to measure the amount of identification a user has toward a product. Pragmatic and hedonic dimensions are independent from each other, but nevertheless share a balanced impact on the overall judgment. The two aspects contribute equally to the overall judgment of the situation/ product and are referred to as Hedonic

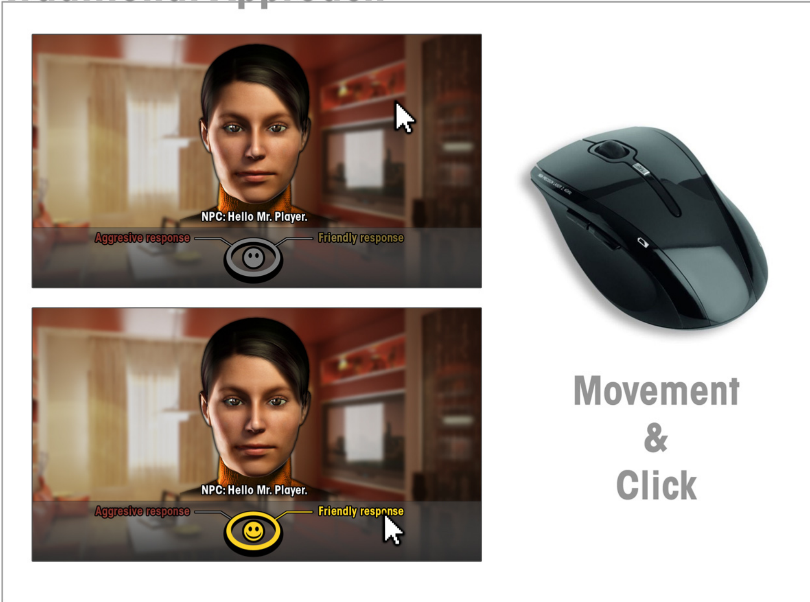


Fig. 3. Traditional approach: by pointing and clicking on the text the player chooses one out of two options

Quality (HQ). Attractiveness (ATT) resembles an overall judgment based on the perceived aesthetic quality.

3.2 Procedure

The study was conducted at the games lab (glab) of the FH Hagenberg with 14 participants, 8 male and 6 female. The age of participants ranged from 22 to 30 (mean=24,54). The procedure took between 15 to 20 minutes per participant. The experimental setting was made up of a MacBook Pro (with a built-in webcam). As a first step, the experimenter welcomed the participant and provided a short introduction text that gave an overview on the procedure and purpose of the study. The evaluation was divided into two major parts: the evaluation of the affective approach and the traditional approach. The order in which participants were confronted with the prototypes was randomized.

Each evaluation part started with a short tutorial showing the basic means of interaction. Afterwards subjects had the possibility to try out the presented approach by themselves. Participants were informed that they had as much time as they desired to interact with the interface prototypes. Hereinafter they were given a short description of the premise that had led to current dialogue situation (one out of two descriptions). When subjects confirmed that every aspect was clear to them the evaluation of the approach began. To avoid material

artifacts, the gender of NPCs in both approaches was also randomized. When the interaction was completed, the experimenter instructed the participants to fill out the online AttrakDiff questionnaire that also contained questions about age, gender, etc. After participants completed the first task, the second approach was presented and evaluated.

4 Results

From a general perspective the results indicate that the use of explicit facial expressions in the context of game dialogue appears to be quite promising but leave room for improvement (see Figure 4). The results were gathered by calculating and summarizing the mean values of the bipolar items (which refer to the different factors like HQ-S or ATT). For a description of the various factors and dimensions see Subsection 3.1 or refer to Hassenzahl et al. [6] to obtain more detailed information. Afterwards paired t-tests were carried out to investigate differences between the traditional and the affective approach.

When looking at HQ values, the affective approach was perceived more positively: the affective approach (HQ: mean=1.23 σ=0.33) , the traditional approach (HQ: mean=-0.67 σ=0.42). Both HQ-I (mean=0.89 σ=0.45) and HQ-S (mean=1.57 σ=0.30) were rated higher in the affective approach than in the traditional concept (HQ-I: mean=-0.18 σ=0.44, HQ-S: mean=-1.15 σ=0.51). Paired t-tests were carried out which showed a significant difference between the HQ-I ($t = 5.79$, $p < .05$) and HQ-S ($t = 13.27$, $p < .05$) of the two different approaches. In general, participants perceived the affective input method as more human oriented than machine oriented. Some subjects reported that they were more concerned about the feelings of the NPC when facial expressions were used as input. This effect can also be seen in the right illustration of Figure 4, which indicates that the affective approach is self oriented.

However, some problems occurred on the PQ level. The affective concept was rated lower (PQ mean=0.35 σ=0.55) than the traditional approach (PQ mean=1.74 σ=0.22). Paired t-tests revealed a significant difference between the affective and the traditional approach (PQ: $t = -5.64$, $p < .05$).

Although the emotion joy was captured without any problem, issues arose when using the anger expression. In some cases it took subjects several minutes to perform a facial expression that was recognized as anger. A second factor that had an impact on the PQ values can be noticed by the observation that most subjects had (a considerable amount of) experience with Role-Playing Games (RPGs) and were familiar with the traditional method. Furthermore, one of the subjects noted that the interaction method might be more applicable for certain game situations (critical situations), since the interaction via a mouse is more effective. This issue, however, can be solved by increasing the facial recognition and reducing the capture time. Material effects (male and female) were avoided since ATT ratings show similar values (affective ATT: mean=0.73 σ=0.55, traditional ATT: mean=0.49 σ=0.41). Paired t-tests showed no significant differences (PQ: $t = 1.26$, $p = .21$).

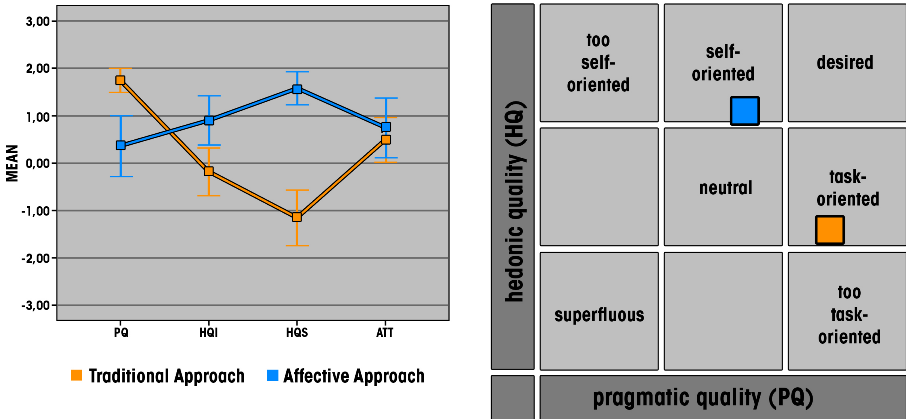


Fig. 4. Resulting diagrams from the evaluation via the AttrakDiff measurement tool - Left figure: summary of the mean values of the individual factors; the affective approach has higher HQ-I and HQ-S values, but lower PQ values in comparison to the traditional concept. ATT values show only minor differences. Right figure: summary of the HQ and PQ dimensions - the traditional addresses usability related aspects, but lacks hedonic features; the affective approach is strongly self-orientated but requires tool related features.

5 Discussion and Outlook

This work introduced a design approach that utilizes facial expressions as an explicit input method for game dialogue systems. In contrast to traditional approaches in game dialogue systems, the proposed approach includes an affective component to reduce misunderstanding of the provided information. With this concept players are encouraged to utilize an additional communication channel. A study was carried out that revealed that the proposed design fosters hedonic aspects in regard to the UX. Subjects gave the affective approach higher HQ-S and HQ-I ratings than the traditional concept. When observing the pragmatic qualities of the system, improvements have to be made. Two areas of improvement are facial expression recognition and capture time. Overall, the affective approach offers new directions in research. A next step in our research will be to add other basic emotions and more available options to interact with an NPC. Furthermore, we are planning to combine traditional and affective methods in order to employ explicit affective input in specific game situations.

Acknowledgement. We would like to thank the Fraunhofer Institute for Integrated Circuits (IIS) for providing the “Realtime FaceDetector”.

References

1. Allwood, J.: On dialogue cohesion. In: Heltoft, L. (ed.) Papers from the Thirteenth Scandinavian Conference of Linguistics, vol. 65. Göteborg University, Department of Linguistics, Roskilde University Centre (1992)

2. Bernhaupt, R., Boldt, A., Mirlacher, T., Wilfinger, D., Tscheligi, M.: Using emotion in games: emotional flowers. In: ACE 2007: Proceedings of the International Conference on Advances in Computer Entertainment Technology, pp. 41–48. ACM Press, New York (2007)
3. Brusk, J., Bjork, S.: Gameplay design patterns for game dialogues. In: Barry, A., Helen, K., Tanya, K. (eds.) Breaking New Ground: Innovation in Games, Play, Practice and Theory: Proceedings of the 2009 Digital Games Research Association Conference. Brunel University, London (2009)
4. Cavazza, M., Pizzi, D., Charles, F., Vogt, T., André, E.: Emotional input for character-based interactive storytelling. In: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, vol. 1, pp. 313–320. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2009)
5. Fraunhofer IIS: Face and object detection (2010), http://www.iis.fraunhofer.de/en/bf/bv/ks/gpe/index.jsp
6. Hassenzahl, M., Burmester, M., Koller, F.: Attrakdiff: Ein fragebogen zur messung wahrgenommener hedonischer und pragmatischer qualität. Mensch und Computer 2003. Interaktion in Bewegung, 187–196 (2003)
7. Hudlicka, E.: Affective game engines: motivation and requirements. In: Proceedings of the 4th International Conference on Foundations of Digital Games, FDG 2009, pp. 299–306. ACM, New York (2009)
8. Kueblbeck, C., Ernst, A.: Face detection and tracking in video sequences using the modified census transformation. Journal on Image and Vision Computing 24(6), 564–572 (2006)
9. Lankes, M., Riegler, S., Weiss, A., Mirlacher, T., Pirker, M., Scherndl, T., Tscheligi, M.: Facial expressions as game input with different emotional feedback conditions. In: ACE 2008: Proceedings of the 2008 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology. ACM Press, New York (2008)
10. Nacke, L.E.: Wiimote vs. controller: electroencephalographic measurement of affective gameplay interaction. In: Proceedings of the International Academic Conference on the Future of Game Design and Technology, Futureplay 2010, pp. 159–166. ACM, New York (2010)
11. Pelachaud, C., Badler, N.I., Steedman, M.: Generating facial expressions for speech. Cognitive Science 20(1), 1–46 (1996)
12. Zhan, C., Li, W., Safaei, F., Ogunbona, P.: Emotional states control for on-line game avatars. In: Proceedings of the 6th ACM SIGCOMM Workshop on Network and System Support for Games, NetGames 2007, pp. 31–36. ACM, New York (2007)

Generating Believable Virtual Characters Using Behavior Capture and Hidden Markov Models

Richard Zhao and Duane Szafron

Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{rxzhao,dszafron}@ualberta.ca

Abstract. We propose a method of generating natural-looking behaviors for virtual characters using a data-driven method called *behavior capture*. We describe the techniques for capturing trainer-generated traces, for generalizing these traces, and for using the traces to generate behaviors during game-play. Hidden Markov Models (HMMs) are used as one of the generalization techniques for behavior generation. We compared our proposed method to other existing methods by creating a scene with a set of six variations in a computer game, each using a different method for behavior generation, including our proposed method. We conducted a study in which participants watched the variations and ranked them according to a set of criteria for evaluating behaviors. The study showed that behavior capture is a viable alternative to existing manual scripting methods and that HMMs produced the most highly ranked variation with respect to overall believability.

1 Introduction

A story-based computer game contains virtual characters. Most of them are AI-controlled non-player characters (NPCs), who interact with the player character (PC), other NPCs, and the environment. Although games display increasingly realistic graphics and physics, NPC behaviors have improved slowly. Players are demanding more realistic behaviors for the NPCs. Rather than standing or wandering aimlessly, NPCs should converse with other NPCs and interact with game objects in realistic ways. They should also react to events such as explosions or unusual events. Creating natural-looking behaviors for NPCs is not inexpensive. In a typical commercial story-based game, there are hundreds or sometimes over a thousand NPCs. Since manually scripting each NPC individually requires extensive resources, most NPCs in most commercial games have staightforward and repetitive behaviors.

We propose a new method of creating NPC behaviors called *behavior capture*, based on the concept of motion capture. With motion capture [6], sensors are attached to the bodies of actors, and as the actors move their bodies, the spatial locations of their body parts are recorded. The data is used to animate virtual characters to move in the same way. Our system of behavior capture is based on a similar idea of using captured traces to guide NPC behaviors, but behavior capture is not a generalization

H.J. van den Herik and A. Plaat (Eds.): ACG 2011, LNCS 7168, pp. 342–353, 2012.
© Springer-Verlag Berlin Heidelberg 2012

of motion capture. Our behavior traces represent high-level intentions as opposed to motion trajectories in space. For example, the NPC may use an entirely different path to approach another NPC during game-play, since the relative positions of the two NPCs may differ from their relative positions during training.

Behavior capture enables a game designer to take control of a particular NPC during training and perform *exemplar* behaviors. It captures traces of the exemplar behaviors and generalizes them. Generalization is necessary to generate natural behaviors with short training times. First, interactions should be generalized. If a particular NPC should talk to a particular set of NPCs (such as rich NPCs only) in a tavern, then the trainer should be able to train this NPC by talking to only one rich NPC. If another NPC should have the same behavior, the trainer should not have to train the second NPC separately. Second, during game play, the NPC should not repeatedly follow the exact training trace. We present several trace generalization mechanisms, including a technique that learns a Hidden-Markov Model (HMM). Our generalization and learning do not require scripting.

The term *behavior capture* has been used in commercial software to describe the LiveAI technique introduced by AiLive [1], and by TruSoft [13]. However, there are no publications describing what technique is used to generalize behaviors after training and there is no indication of the level of behaviors that can be learned, although there is a video that highlights some behavior in a showcase combat scenario [2].

To verify the utility of behavior capture, we created a tavern scene in a commercial video game. We generated a set of scene variants, using a collection of behavior generation techniques, including manual scripting and several forms of behavior capture. We conducted a study in which participants *played* each scene variant and ranked them by: (1) most active characters, (2) most unpredictable characters, (3) most plausible action sequences, (4) most diverse character actions. Participants also ranked overall believability and rated overall believability of each variant.

2 Related Work

There are many methods of creating behaviors for virtual characters. Traditionally, programmers had to script individual behaviors for each character. Orkin and Roy devised a data-driven approach to generating behaviors, using unsupervised learning of behavior and dialogue in a game that simulates a restaurant [10]. They take advantage of the massive online-gaming community and use it to collect data as training examples. A character in the game has a set of goals and corresponding priorities, used to guide interactions. After goal selection, the character retrieves candidate plans and sends them to the critics system, which uses criteria to reject plans. Experiments compare the ability of the plan network and humans to differentiate between typical and atypical restaurant behavior. Thurau et al. [12] describes methods of inferring goals from replays of human games for a virtual character in a First-Person Shooter game. Ontanon et al. [9] proposes a planning system for a Real-Time Strategy game that can be learned from human demonstration. These two approaches assume there are specific goals (such as killing an enemy) for

the virtual characters to achieve, and different methods are used to find ways to satisfy such goals. Our research targets day-to-day behaviors of virtual characters in a non-combat virtual environment, where NPCs often behave without clear-cut goals.

MacNamee [7] proposed a technique called role-passing, which allows an NPC to exhibit behavior variations by assuming roles in particular situations. Such NPCs can be implemented using *level-of-detail*, which means that their behaviors can be modeled abstractly when the player is not looking, and in more detail when the player is focused on them. These *proactive persistent agents* provide a realistic experience. Individual behaviors are driven by an artificial neural network, which is trained using about 300 hand-crafted examples by the author. While these examples contained a set of interesting behaviors, their flexibility to adapt to other situations has not been demonstrated. It is not clear how a game designer can add additional behaviors not included in the pre-designed training examples. Research has been done in the past to analyze player statistics extracted from in-game traces or character attributes [8] [11]. We want to provide a better gaming experience for players, specifically by enabling game designers to create more interesting NPC behaviors.

3 Capturing Behaviors

Behavior capture is a data-driven approach to generating virtual character behaviors. Instead of a programmer specifying how each character should move, speak, and interact with the environment, a game designer takes control of the character and performs the actions that the designer would like to see this character perform. This is done in a game session called *training mode*. In training mode, the designer-controlled character actions are recorded with a behavior capture system. The system remembers what each character did and uses this data to generate new behaviors during game play. There is no need to write programming scripts.

3.1 Training in Neverwinter Nights

We constructed a prototype of our behavior capture system in BioWare Corp.'s Neverwinter Nights (NWN). NWN includes a simple-to-use Toolset that allows designers to create new stories using a C-like scripting language. Since NWN is a medieval fantasy game, tavern scenes are common. A tavern typically has patrons, a bartender, and sometimes entertaining bards. We created a tavern scene and used it as a test-bed for our behavior capture system (Figure 1).

In training mode, the characters start with no behaviors. A game designer takes control of a *trainee* character. Figure 2 shows a trainee in the center of the screen. At the bottom of the screen, there are buttons representing the actions a trainee can perform (pressing modifier keys display other sets of actions). If the designer clicks on the *Face* button and then clicks on another character, the trainee will turn and face the clicked character, and the *Face* action will be recorded as an action in a sequence of actions the trainee should perform. The game designer can switch trainees at any time, by clicking on the *Become* button and clicking on another character. If a trainer

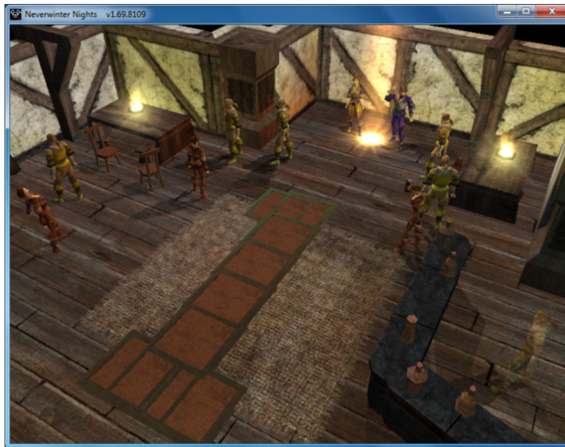


Fig. 1. The tavern scene in Neverwinter Nights

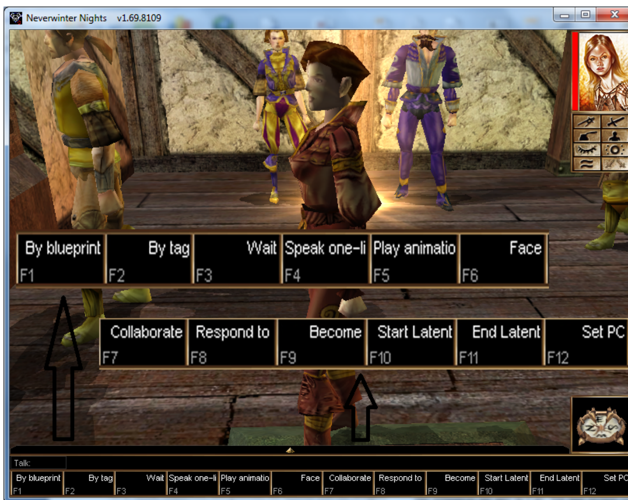


Fig. 2. Training a character. The action bar is enlarged in the figure for clarity.

makes an error, the offending trace can be deleted from the training record. To avoid pressure on the designer during training, designer pauses are ignored in training mode and if the trainer wants the trainee to pause, an explicit wait action is selected.

3.2 Behavior Types

Cutumisu [4] introduced the behavior ontology shown in Figure 3. Behaviors are categorized as independent or collaborative. Independent behaviors are performed by one NPC, while collaborative behaviors are performed with a partner. An NPC may have an independent behavior to *sit on a chair* and a collaborative behavior to *talk to another NPC*. Behaviors can also be classified as proactive, reactive (reacting to a partner's initiative), or latent.

A new proactive behavior is initiated when no other behavior is active. A latent behavior is triggered by a specific event and has a higher priority than any proactive behavior so it can interrupt. For example, an NPC may perform a proactive behavior to *sit* on a chair. When the NPC is done, a new proactive behavior is selected. The NPC may *wait* or *talk* to another NPC. However, if an explosion occurs, a latent behavior to flee from the explosion can be triggered and the *flee* behavior will interrupt the current proactive behavior.

Our behavior capture system supports this behavior ontology during training mode. For example, to train a collaborative behavior, a designer clicks on the *Collaborate* button and clicks on another character so the system recognizes the collaboration partner. The designer trains one character first, and then switches to the other character and trains a corresponding reactive behavior, clicking the *Collaborate* button again to end the collaborative behavior trace.

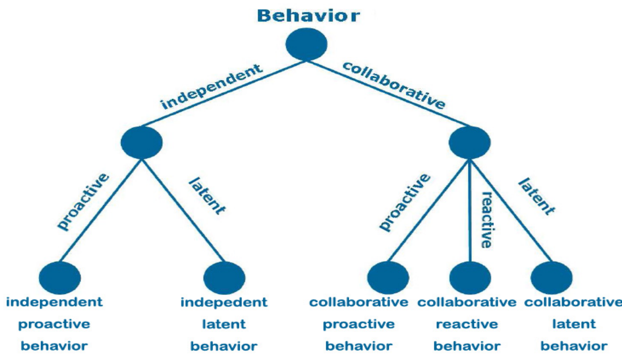


Fig. 3. Behavior architecture, adapted from Cutumisu [4]

A latent behavior is based on a game event. To train a latent behavior, the trainer first clicks on the *Start Latent* button. The designer next triggers the appropriate event. In our tavern scene, the designer may want some tavern patrons to cheer when the bard finishes performing and *exits* the stage. Since *exits a trigger* (an area on the ground) is an event in the game, performing this action in *Latent* mode enables the behavior capture system to record this event. The trainer clicks on the bard and then moves the bard out of the trigger area. Once the event is recorded, the designer trains an NPC to react to this event by clicking on the appropriate NPC and selecting behaviors, such as *face* the bard behavior and a *cheer* animation behavior.

4 Generating Behaviors

After data is gathered using the training mode, the behavior capture system produces the NPC behaviors. Currently, the behavior capture system supports three types of generalizations. First, there can be hundreds of NPCs, and the designer may want the training of one NPC to apply to multiple NPCs – *character generalization*. Second, the designer may want a trained NPC interaction with a specific object to generalize to an interaction with any one of a group of objects – *object generalization*. Third, the

designer may not want the NPC to perform the training actions strictly in the training sequence order during game play and then repeat them in the same order once the sequence is complete, so we perform *sequence generalization*.

4.1 Character and Object Generalization

To support character and object generalization, the behavior capture system uses *categories* of objects and characters. For example, if a designer trains a character to *sit on a chair*, the action is not to *sit on that specific chair*, but to *sit on any chair in a category*. During game play, the character would sit on one of many chairs. To force a character to sit on a specific chair, the designer places this chair in its own category. Object categorization mechanisms are game-specific. For example, in NWN, the designer can use two different categorization mechanisms – tags and blueprints. The designer assigns a tag to each game object. The same tag can be assigned to different kinds of objects. During training, any interaction with an object can be generalized to an interaction with a random object with the same tag. For example, the trainer can train an NPC to converse with any tavern patron whose tag is *conversable* by conversing with any one of them. Alternatively, in NWN, the trainer can choose to generalize by blueprint – the template used to create an object. In this case, sitting on a chair would train an NPC to sit on any object created using the chair blueprint, regardless of tags. In our NWN trainer, the designer can toggle between using tags or blueprints to categorize objects. To support character generalization, when a designer trains an NPC for one blueprint, all NPCs with the same blueprint receive this training. It is easy to make custom blueprints from existing blueprints so creating categories that correspond to groups with common behaviors is straightforward.

4.2 Sequence Generalization

A behavior capture system needs an algorithm to order behaviors based on the training traces. A straightforward approach, *no sequence generalization,* generates a sequence of actions that exactly matches the recorded sequence and then repeats this sequence. However, a player may regard this repetition as unnatural. Alternative approaches could select actions from the set of training actions in a non-deterministic manner. For example, the system could sample uniformly from the set of all trained actions using *random action sequence generalization*. However, in many situations the order is important. To provide some designer control over the non-determinism, we divide the training actions for a single NPC into a set of traces of actions. A designer starts a trace, performs a sequence of actions and ends the trace. The designer usually performs many traces, each of which contains a short sequence of actions that form a cohesive sequence. For example, one trace may consist of three actions: (1) *wait a few seconds*, (2) *say "I'm thirsty"* and (3) *walk to the bar*. Another trace may consist of three actions: (1) *wait a few seconds*, (2) *say "I'm tired"*, and (3) *walk to a chair*. The random action technique could result in unrealistic action sequences, such as (1) *say "I'm tired"*, (2) *walk to the bar* or (1) *say "I'm thirsty"*, (2) *walk to a chair* or even (1) *wait a few seconds*, (2) *wait a few seconds*, (3) *wait a few seconds*, etc.

The *random trace sequence generalization* technique uniformly select traces instead of actions. It tries to maintain the plausibility of action sequences created by the designer. However, over longer periods of time (many traces), this technique can produce behaviors that players view as repetitious. Therefore, we created a third sequence generalization that maintains traces to some extent, while producing emergent sequences that may reduce repeatability. We introduce *HMM sequence generalization* that uses a Hidden Markov Model (HMM) to select actions that have a bias towards selecting actions in the order specified by the designer.

A Markov Model is a statistical model with states, transitions, and outputs. One state is a special state called the start state. Each state is connected to a set of other states by probabilistic transitions. In addition, each state has an output probability of outputting a set of outputs. An HMM is a Markov Model of which the states are unobserved (hidden). In our application, the output is one of the behavior actions that the designer used in a trace. Hidden states are hidden to a game designer. The number of hidden states is a parameter of the HMM sequence generalization technique.

One HMM generates the proactive behaviors for each character. The Baum–Welch Algorithm [3], a generalized expectation-maximization algorithm, uses the training traces to teach the HMM. The HMM adjusts its transition and output probabilities to fit the trace sequences. If the trainer follows the *say "I'm thirsty"* action with the *walk to the bar* action, the HMM will favor this action order. If the trainer trains the NPC to converse three times as often as ordering a drink, the HMM will generate a similar 3 to 1 ratio. The behaviors generated by the HMM are stochastic, but are somewhat consistent with the training traces. The number of hidden states parameters controls the consistency; a higher consistency is achieved by more hidden states.

5 User-Study and Evaluation

5.1 User-Study

We conducted a user study to evaluate the utility of behavior capture. Participants were enrolled in a first-year university psychology class. They did not necessarily have video game experience. Our goal was to show that behavior capture is a viable alternative to typical commercial game NPC behaviors created by manual scripting. We created six variants of a tavern scene in NWN, which are identical in all aspects except in the way that the NPC behaviors were generated. The six scene variations were constructed using the techniques listed in Table 1. The mapping between Scene Variation number and technique was generated randomly to avoid any bias.

Study participants were asked to watch the six variations, and to rank them according to the criteria listed as the first column of Table 2. We also asked the participants to rate the overall believability of each variation on a scale of 1-4.

Technique T1 is a baseline, with all characters exhibiting only stock idling animations provided by the NWN game engine, such as stretching their arms. Technique T2 is hand-scripted – characters behave according to a representative commercial role-playing game, Dragon Age: Origins. The variation was scripted to combine the behaviors in two taverns, *Lothering* (bards entertaining) and *Redcliffe* (a server who walks around). If we only used one of the taverns the behaviors would have been quite straightforward and we believe the evaluations of this variation would have resulted in a worse ranking. The other four techniques used behavior capture.

Table 1. Behavior generation techniques. Scene Variation numbers (shown to participants instead of technique numbers) were randomly assigned to techniques to avoid bias

Technique	Scene Variation Number	Behavior Generation
T1	4	No behaviors added (idle)
T2	6	Behaviors hand-scripted by a programmer
T3	5	Behavior capture with no sequence generalization
T4	1	Behavior capture with random action sequence generalization
T5	3	Behavior capture with random trace sequence generalization
T6	2	Behavior capture with HMM sequence generalization (using 8 hidden states)

For behavior capture, fourteen different proactive actions (behaviors) were used:

a1) Wait 5 seconds
a2) Collaborate with another patron by conversing
a3) Wave at another patron
a4) Face another patron
a5) Walk to a chair
a6) Walk to the bar
a7) Speak "I'm thirsty"
a8) Speak "I'm tired"
a9) Speak "I'm lonely"
a10) Speak "See you later"
a11) Speak "I'm bored"
a12) Speak "The inn-keeper is busy"
a13) Speak "I see a friend"
a14) Speak "What was that noise?"

The collaborative *converse* behavior included 4 tasks: *face the collaborator*, *walk to the collaborator*, *speak* and then *listen*. The training set contained 10 traces of three actions each: [a5, a7, a6], [a1, a4, a3], [a6, a8, a5], [a9, a2, a10], [a4, a1, a4], [a5, a11, a6], [a6, a12, a5], [a13, a2, a10], [a14, a4, a4], [a4, a3, a1].

Technique T3 is behavior capture with no sequence generalization. The next action is picked directly from the trainer's traces, combining traces into a single long trace to eliminate randomness in picking actions. The list of actions in order was [a5, a7, a6, a1, a4, a3, a6, a8, a5, …, a4, a3, a1] and this list was repeated forever.

Technique T4 is behavior capture with random action sequence generalization. An action is selected randomly, ignoring training traces. The next action to perform is: `random{a1, a2, … , a14}`.

Technique T5 is behavior capture with random trace sequence generalization. It picks a random trace from available training traces, and performs the actions in the chosen trace order. Therefore the next three actions in order are: `random{[a5, a7,a6],[a1,a4,a3],[a6,a8,a5],…,[a4,a3,a1]}`. Technique T5 is just one of the parameterized HMM generalizations. Since an HMM remembers its

previous state information through the transitional probabilities between pairs of hidden states, with enough hidden states, the HMM can remember the exact order of the training traces and reproduce actions in the same order as the training traces. Technique T6 is behavior capture with HMM sequence generalization and it uses an HMM with eight hidden states.

In addition to the proactive behaviors, each patron was trained to perform one reactive behavior: face the collaborator, walk to the collaborator, listen and then speak. This behavior is used to respond to the proactive collaborative converse behavior. Patron training also included one latent behavior: *when the bard exits the stage, face the bard and cheer*. A multi-queue behavior architecture was used so that the latent behavior would interrupt any proactive behavior and the interrupted behavior would be resumed after the latent behavior was completed [5]. The bard was trained with two proactive behaviors. The first was: *sing*, *wait*, and *leave the stage*. The second was: *return to the stage*.

We had two main hypotheses. First, that behavior capture would rank higher than the *no behavior* and *manually scripted behavior* techniques. Second, that sequence generalization would be a factor in the perception of believable characters. Specifically, the order of behavior capture rankings would be: HMM, random traces, random actions, and no sequence generalization.

5.2 Preliminary User-Study

We conducted a preliminary user study to evaluate the effects of the number of training traces. We wanted to determine how the number of traces would influence user perceptions. The goal was to determine if users could distinguish between quite a small number of training traces (4) and a larger number (9). The user study was constructed similarly to the main user study that is described in the previous section, except that scene variations with 4 training traces were compared to scene variations with 9 training traces. The results show that with 95% confidence the variations with higher numbers of traces produced more believable scenes. This is an expected trade-off between quality and workload. Based on this result, we set the number of training traces in our main user-study closer to the higher number (we selected 10 traces).

5.3 Results

In the main user study, there were 27 participants. However, a few participants did not answer carefully enough for their responses to be considered valid, with some participants not answering some questions and some participants providing what seemed to be random answers (e.g., ranking 1,2,3,4,5,6 for all criteria). Therefore, the results of each questionnaire were validated for self-consistency. One question asked the respondent to rank the six variations according to overall believability, while another question asked the respondent to rate the six variations individually on a scale of 1 to 4 according to overall believability. To ensure that the participants answered the questions carefully, we removed a questionnaire if the rankings and ratings contained more than one inconsistency between the rating and ranking questions. After this consistency check, a total of 21 valid questionnaires remained.

Table 2 shows the average technique rankings for the 6 techniques. Each number represents the average ranking for the particular technique for the particular criteria

over the 21 responses. For example, technique T6 is ranked 4.29 on average (where 6 is the highest ranking) among the 6 techniques, according to the criteria *active characters*. The results show that in general, technique T6 is ranked highly for all the criteria except *unpredictable characters*. Note that high rankings are better than low ones for all criteria except *unpredictable characters* and that the *overall believability* is not an average of the other criteria. It was a separate question on the survey.

Table 2. Average Technique Ranking (6 is Highest, 1 is Lowest) and Average Overall Rating (out of 4). Higher numbers are better in all criteria except *unpredictable characters*. Standard deviations are shown in parentheses.

Criteria	T1	T2	T3	T4	T5	T6
active characters (ranking)	1.19 (0.60)	3.38 (1.72)	3.90 (1.67)	4.29 (1.23)	3.95 (1.53)	4.29 (1.06)
unpredictable characters (ranking)	3.00 (2.24)	3.29 (1.65)	4.48 (1.60)	3.52 (1.50)	3.62 (1.60)	3.10 (1.34)
plausible sequences (ranking)	2.00 (1.45)	2.38 (1.63)	3.57 (1.57)	4.38 (1.40)	4.29 (1.35)	4.38 (1.20)
diverse actions (ranking)	1.24 (0.54)	3.29 (1.68)	4.10 (1.67)	3.86 (0.96)	3.81 (1.66)	4.71 (1.10)
overall believability (ranking)	1.33 (0.80)	3.05 (1.69)	3.67 (1.28)	4.00 (1.26)	4.10 (1.61)	4.86 (1.15)
overall believability (rating)	1.20 (0.41)	2.05 (1.16)	2.19 (0.87)	2.71 (0.90)	2.57 (0.87)	2.85 (0.59)

We used a Friedman statistical test, and compared each row of Table 2 to avoid the alpha-inflation effect. It indicated that there are significant differences in the average rankings of the six techniques for each criterion at the 95% confidence level. We used a Friedman test instead of ANOVA because of the ranked data. Based on the positive result of the Friedman test, T-tests were used to compare pairs of rankings.

Table 3 shows the p-values of subsequent T-tests between the average rankings of technique T6 versus each of the other techniques. The most obvious result is that T6 was ranked significantly higher than T1 and T2 in all aspects except unpredictability. This study indicates that hand-scripted characters are perceived as less diverse, less plausible and have less active characters than character behaviors generated using behavior capture with HMM sequence generalization. The study also shows that T6 ranked significantly higher than all other tested techniques for overall believability.

It is perhaps surprising that T6 was perceived as significantly better for overall believability compared to T3, T4, and T5, even though T6 was not perceived as significantly better on some of the four component criteria. This indicates that either there is a missing criterion that is necessary for overall believability or that believability cannot straightforwardly be decomposed into parts based on criteria. This is a crucial question in trying to measure believability, as pointed out by MacNamee [7], who endorsed the evaluation of aspects of believability rather than overall believability to reduce subjectivity. To evaluate the importance of our criteria, we asked participants to rate the importance of each of the four criteria. Table 4 shows the average importance computed from the responses of the study participants. As expected, unpredictability is the least important in the eyes of the participants.

Table 3. p-values (to two decimals) from T-tests on Technique T6 versus each other technique for each criterion. Entries with a dark background are significant at the 95% confidence level

Criteria	T1 vs. T6	T2 vs. T6	T3 vs. T6	T4 vs. T6	T5 vs. T6
Active	0.00	0.05	0.22	0.50	0.20
Unpredictable	0.44	0.37	0.01	0.09	0.14
Plausible	0.00	0.00	0.04	0.50	0.40
Diverse	0.00	0.00	0.13	0.00	0.02
Overall	0.00	0.00	0.00	0.03	0.03

Table 4. The average importance of the four criteria. Participants rated each criteria on a scale of -3 to 3. A positive number means important in contributing positively to overall believability. A negative number means important in contributing negatively. The larger the absolute value the more important it is.

Criteria	Average Importance
active	1.76
unpredictable	0.71
plausible	2.19
diverse	1.52

We will conduct studies on different variations and a larger participant pool to increase confidence that behavior capture generates significantly better behaviors.

6 Conclusion

The video game industry continues to grow and game designers are becoming more specialized in their own areas. In addition, recent story-based video games have started providing tools so that non-professionals can design their own stories. However, in order to use these tools successfully, a designer needs to know programming, since characters in game are controlled by programmed scripts.

In this paper we propose a new tool for game designers that allow them to create behaviors for virtual characters, without having to learn complicated programming. Using an analogy to motion capture, we propose a data-driven method of creating behaviors for NPCs by behavior capture. Game designers take control of a particular NPC and perform the desired behaviors for this NPC. Using category-based generalization for characters and objects, and Hidden-Markov Models for sequence generalization, our behavior capture technique produces a variety of behaviors learned from the training traces. We performed a user study to confirm that behavior capture produces behaviors that are perceived as significantly superior with regards to character activity level, unpredictability and behavior diversity compared to the

scripted behaviors seen in a typical commercial story-based game. This study also showed that using HMMs for sequence generalization, instead of raw behavior traces contributes significantly to perceived overall believability. For future work, larger user studies with more participants and different scenes should be conducted to provide move evidence of the utility of behavior capture and more insight into the best technique for sequence generalization.

References

1. AiLive, http://www.ailive.net/
2. AiLive. AiLive LiveCombat: Artificial intelligence and behavior capture for games, http://www.youtube.com/watch?v=u8oNTLzCFNU
3. Baum, L.E., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. Ann. Math. Statist. 41(1), 164–171 (1970)
4. Cutumisu, M.: Using Behavior Patterns to Generate Scripts for Computer Role-Playing Games. Ph.D. thesis. University of Alberta (2010)
5. Cutumisu, M., Szafron, D.: An Architecture for Game Behavior AI: Behavior Multi-Queues. In: Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE), Stanford, USA, pp. 20–27 (2009)
6. Gleicher, M.: Animation from observation: Motion capture and motion editing. ACM SIGGRAPH Computer Graphics 33, 4 (2000)
7. MacNamee, B.: Proactive Persistent Agents: Using Situational Intelligence to Create Support characters in Character-Centric Computer Games. PhD Thesis. Trinity College Dublin (2004)
8. Mahlmann, T., Drachen, A., Togelius, J., Canossa, A., Yannakakis, G.N.: Predicting Player Behavior in Tomb Raider: Underworld. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (2010)
9. Ontanon, S., Bonnette, K., Mahindrakar, P., Gomez-Martin, M.A., Long, K., Radhakrishman, S.J., Ram, R.: A Learning from Human Demonstrations for Real-Time Case-Based Planning. In: The IJCAI 2009 Workshop on Learning Structural Knowledge From Observations (2009)
10. Orkin, J., Roy, D.: Automatic Learning and Generation of Social Behavior from Collective Human Gameplay. In: International Conference on Autonomous Agents and Multiagent Systems (2009)
11. Thurau, C., Bauckhage, C.: Analyzing the Evolution of Social Groups in World of Warcraft. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (2010)
12. Thurau, C., Bauckhage, C., Sagerer, G.: Imitation learning at all levels of game-AI. In: Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education, pp. 402–408. University of Wolverhampton (2004)
13. TruSoft: Artificial Contender, http://www.trusoft.com/principles.html

Author Index

Abbadi, Mohamed 320
Althöfer, Ingo 258

Baier, Hendrik 39
Baudiš, Petr 24
Bouzy, Bruno 96
Brausen, Henry 220
Bugliesi, Michele 320

Cazenave, Tristan 196
Chang, Chieh-Min 121
Chang, Hung-Jui 306
Chen, Jitong 108
Chou, Cheng-Wei 84
Chou, Ping-Chiang 84
Costantini, Giulia 320
Coulom, Rémi 146

Doghmen, Hassen 84

Ederer, Thorsten 270

Gailly, Jean-loup 24
Ganzfried, Sam 282

Hashimoto, Junichi 1
Haworth, Guy McC. 230, 244
Hayward, Ryan B. 220
Hnath, Steven 170
Hoki, Kunihito 158, 184
Hsu, Tsan-sheng 306

Ikeda, Kokolo 1

Jouandeau, Nicolas 196

Kaneko, Tomoyuki 158, 184, 208
Kishimoto, Akihiro 1

Lankes, Michael 333
Lee, Chang-Shing 84
Li, Shiyuan 108
Lin, Hung-Hsuan 121
Lin, Ping-Hung 121
Lin, Yi-Shan 121

Lorentz, Richard J. 52
Lorenz, Ulf 270
Louveaux, Quentin 295
Lv, Huizhan 108

Macieja, Bartlomiej 230
Maggiore, Giuseppe 320
Métivier, Marc 96
Mirlacher, Thomas 333
Müller, Martin 13, 220

Neller, Todd W. 170
Nijssen, J. (Pim) A.M. 72

Opfer, Thomas 270
Orsini, Renzo 320

Papahristou, Nikolaos 134
Pellier, Damien 96

Qadir, Abdul 220

Refanidis, Ioannis 134
Regan, Kenneth W. 230
Rusz, Á. 244

Saffidine, Abdallah 196
Spanò, Alvise 320
Spies, David 220
Stankiewicz, Jan A. 60
St-Pierre, David L. 295
Su, Tsan-Cheng 84
Szafron, Duane 342

Tanaka, Tetsuro 208
Teytaud, Fabien 84
Teytaud, Olivier 84, 295
Tsai, Hsin-Ti 121
Tsai, Meng-Tsung 306

Uiterwijk, Jos W.H.M. 60

Van Eyck, Gabriel 13

Wang, Hui-Ming 84
Wang, Jiao 108
Wang, Mei-Hui 84
Wei, Xin 108

Winands, Mark H.M. 39, 60, 72
Wolf, Jan 270
Wu, I-Chen 121
Wu, Li-Wen 84

Xu, Xinhe 108

Yamaguchi, Kazunori 208
Yamaguchi, Yoshiaki 208
Yen, Shi-Jim 84
Yoshizoe, Kazuki 1

Zhao, Richard 342