H. Jaap van den Herik
Xinhe Xu
Zongmin Ma
Mark H. M. Winands (Eds.)

LNCS 5131

Computers and Games

6th International Conference, CG 2008
Beijing, China, September/October 2008
Proceedings

ifip

Lecture Notes in Computer Science 5131

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

H. Jaap van den Herik Xinhe Xu
Zongmin Ma Mark H. M. Winands (Eds.)

Computers and Games

6th International Conference, CG 2008
Beijing, China, September 29 - October 1, 2008
Proceedings

Springer

Volume Editors

H. Jaap van den Herik
Tilburg centre for Creative Computing (TiCC)
Tilburg University
Tilburg, The Netherlands
E-mail: H.J.vdnHerik@uvt.nl

Xinhe Xu
College of Information Science and Engineering
Northeastern University
Shenyang, China
E-mail: xuxinhe@ise.neu.edu.cn

Zongmin Ma
College of Information Science and Engineering
Northeastern University
Shenyang, China
E-mail: mazongmin@ise.neu.edu.cn

Mark H. M. Winands
Maastricht ICT Competence Centre (MICC)
Maastricht University
Maastricht, The Netherlands
E-mail: m.winands@micc.unimaas.nl

Library of Congress Control Number: 2008934730

CR Subject Classification (1998): G, I.2.1, I.2.6, I.2.8, F.2, E.1

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

ISSN 0302-9743
ISBN-10 3-540-87607-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-87607-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12517534 06/3180 5 4 3 2 1 0

Preface

This book contains the papers of the *6th* Computers and Games Conference (CG 2008) held in Beijing, China. The conference took place from September 29 to October 1, 2008 in conjunction with the 13^{th} International Computer Games Championship and the 16^{th} World Computer Chess Championship.

The Computers and Games conference series is a major international forum for researchers and developers interested in all aspects of artificial intelligence and computer game playing. The Beijing conference was definitively characterized by fresh ideas for a large variety of games. Earlier conferences took place in Tsukuba, Japan (1998), Hamamatsu, Japan (2000), Edmonton, Canada, (2002), Ramat-Gan, Israel (2004), and Turin, Italy (2006).

The Programme Committee (PC) received 40 submissions. Each paper was initially sent to at least two referees. If conflicting views on a paper were reported, it was sent to an additional referee. Out of the 40 submissions, one was withdrawn before the final decisions were made. With the help of many referees (listed after the preface), the PC accepted 24 papers for presentation at the conference and publication in these proceedings.

The above-mentioned set of 24 papers covers a wide range of computer games. Twelve of the games are played in practice by human players, viz., Go, Western Chess, Chinese Chess (Xiangqi), Japanese Chess (Shogi), Amazons, Chinese Checkers, Hearts, Hex, Lines of Action, Othello, Siguo, and Spades. Moreover, there was one puzzle, viz., SameGame, and two theoretical games, viz., Synchronized Domineering and multi-player Can't Stop.

The papers deal with many different research topics including cognition, combinatorial game theory, search, knowledge representation, and optimization.

We hope that the readers will enjoy the research efforts of the authors. Below we provide a brief outline of the 24 contributions, in the order in which they are printed in the book.

"Single-Player Monte-Carlo Tree Search," by Maarten Schadd, Mark Winands, Jaap van den Herik, Guillaume Chaslot, and Jos Uiterwijk, proposes a new Monte-Carlo Tree Search variant, called Single-Player Monte-Carlo Tree Search (SP-MCTS). The method is tested on the puzzle SameGame. It gained the highest score so far on the standardized test set of 20 positions.

"Amazons Discover Monte Carlo" is authored by Richard Lorentz. He incorporated the basic ideas of MC/UCT into the Amazons program InvaderMC and then made improvements to create a hybrid MC/UCT program. This hybrid version is considerably stronger than minimax-based Invader.

"Monte-Carlo Tree Search Solver," by Mark Winands, Yngvi Björnsson, and Jahn-Takeshi Saito, investigates the application of MCTS for the game Lines of Action (LOA). A new MCTS variant, called MCTS-Solver, has been designed to improve playing narrow tactical lines in sudden-death games such as LOA.

"An Analysis of UCT in Multi-player Games" is written by Nathan Sturtevant. It provides an analysis of the UCT algorithm in multi-player games, showing that UCT is computing a mixed-strategy equilibrium, as opposed to Max^{n}, which computes a pure-strategy equilibrium. The author shows that UCT performs as well or better than existing algorithms.

"Multi-player Go" by Tristan Cazenave addresses the application of Monte-Carlo Tree Search to multi-player Go. A straightforward and effective heuristic is defined. It is used in the playouts, which models coalitions of players.

"Parallel Monte-Carlo Tree Search" is authored by Guillaume Chaslot, Mark Winands, and Jaap van den Herik. It discusses three parallelization methods for MCTS: *leaf parallelization*, *root parallelization*, and *tree parallelization*. Experiments in 13×13 Go reveal that in the program MANGO root parallelization leads to the best results.

"A Parallel Monte-Carlo Tree Search Algorithm," by Tristan Cazenave and Nicolas Jouandeau, presents a parallel Master-Slave algorithm for Monte-Carlo Tree Search. The algorithm is tested on a network of computers using various configurations.

"Using Artificial Boundaries in the Game of Go," by Ling Zhao and Martin Müller, describes a new general framework for finding boundaries in such a way that existing local search methods can be used. By applying a revised local UCT search method, it is shown experimentally that this framework increases performance on local Go problems with open boundaries.

"A Fast Indexing Method for Monte-Carlo Go," written by Keh-Hsun Chen, Dawei Du, and Peigang Zhang, proposes a direct indexing approach to build and use a complete 3×3 pattern library. Testing results show that their method increases the winning rates of GO INTELLECT against GNU GO on 9×9 games by over 7%, taking the tax on the program speed into consideration.

"An Improved Safety Solver in Go Using Partial Regions" is a contribution by Xiaozhen Niu and Martin Müller. The authors introduce a new technique that is able to prove that parts of large regions are safe. Experimental results show that the new technique significantly improves the performance of their previous state-of-the-art safety-of-territory solver.

"Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength" is written by Rémi Coulom. The author proposes whole-history rating (WHR), a new method to estimate the time-varying strengths of players involved in paired combats. Experiments demonstrate that, in comparison to Elo, Glicko, TrueSkill, and decayed-history algorithms, WHR produces better predictions.

"Frequency Distribution of Contextual Patterns in the Game of Go" is a joint effort by Zhiqing Liu, Qing Dou, and Benjie Lu. They show that the Zipfian frequency distribution of Go patterns in professional games is deliberate by rejecting the null hypothesis that the frequency distribution of patterns in random games exhibits a Zipfian frequency distribution.

"A New Proof-Number Calculation Technique for Proof-Number Search" is a contribution by Kazuki Yoshizoe. The paper proposes a new straightforward

calculation technique for proof numbers. It is called *dynamic widening*. Its performance is tested on capturing problems of Go on 19×19 boards.

"About the Completeness of Depth-First Proof-Number Search," written by Akihiro Kishimoto and Martin Müller, resolves the question of completeness of df-pn: its ability to solve any finite boolean-valued game tree search problem in principle, given unlimited amounts of time and memory. The main results are that df-pn is complete on finite directed acyclic graphs (DAG) but incomplete on finite directed cyclic graphs (DCG).

"Weak Proof-Number Search," by Toru Ueda, Tsuyoshi Hashimoto, Junichi Hashi-moto, and Hiroyuki Iida, introduces a new search idea using proof number and branching factor as search estimators. It is called *weak proof-number search*. The experiments performed in the domain of shogi and Othello show that the proposed search algorithm is potentially more powerful than the original proof-number search and its depth-first variants.

"Cognitive Modeling of Knowledge-Guided Information Acquisition in Games" is written by Reijer Grimbergen. The paper argues that Marvin Minsky's *society of mind* theory is a good candidate for a cognitive theory to define chunks and to explain the relation between chunks and problem-solving tasks. A reproduction experiment is performed in shogi showing that perception is guided by knowledge in long-term memory.

"Knowledge Inferencing on Chinese Chess Endgames" is a contribution by Bo-Nian Chen, Pangfeng Liu, Shun-Chin Hsu, and Tsan-sheng Hsu. They propose a novel strategy that applies a knowledge-inferencing algorithm on a sufficiently small database to determine whether endgames with a certain combination of material are advantageous to a player. Their experimental results show that the performance of the algorithm is good and reliable.

"Learning Positional Features for Annotating Chess Games: A Case Study," by Matej Guid, Martin Možina, Jana Krivec, Aleksander Sadikov, and Ivan Bratko, points out certain differences between the computer programs, which are specialized for playing chess, and their own program, which is aimed at providing quality commentary. Through a case study, the authors present an application of argument-based machine learning in order to provide their annotating system with an ability to comment on various positional intricacies of positions in chess.

"Extended Null-Move Reductions" is a contribution by Omid David-Tabibi and Nathan Netanyahu. The authors review several versions of null-move pruning, and present their enhancement *null-move reductions* (NMR), which allows for a deeper search with greater accuracy. Experimental results using their own chess program, Falcon, show that NMR outperforms the conventional methods. Here we see that the tactical benefits of a deeper search outweigh the deficiencies.

"GTQ: A Language and Tool for Game-Tree Analysis," by Jónheiður Ísleifsdóttir and Yngvi Björnsson, presents the game tree query language (GTQL), a query language specifically designed for analyzing game trees. Moreover, the authors discuss the design and implementation of the game tree query tool (GTQT), a program that allows efficient execution of GTQL queries on game-tree log files.

"Probing the 4-3-2 Edge Template in Hex," written by Philip Henderson and Ryan Hayward, introduces path-domination and neighborhood-domination, two refinements of domination in Hex, and uses these notions to find conditions under which probes of an opponent 4-3-2 edge template are inferior moves that can be ignored in the search for a winning move.

"The Game of Synchronized Domineering" is a contribution by Alessandro Cincotti and Hiroyuki Iida. For the game of Synchronized Domineering, the paper presents the solutions for all the $m \times n$ boards with $m \leq 6$ and $n \leq 6$. The authors also give results for the $n \times 3$ boards, $n \times 5$ boards, and some partial results for the $n \times 2$ boards.

"A Retrograde Approximation Algorithm for Multi-player Can't Stop," by James Glenn, Haw-ren Fang, and Clyde Kruskal, studies the computational solution of multi-player Can't Stop, and presents a retrograde approximation algorithm to solve it by incorporating the multi-dimensional Newton's method with retrograde analysis. Results on small versions of three- and four-player Can't Stop are given.

"AWT: Aspiration with Timer Search Algorithm in Siguo" is a joint effort by Hui Lu and ZhengYou Xia. The paper proposes a modified alpha-beta aspiration search algorithm, which is called alpha-beta aspiration with timer algorithm (AWT).

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 Beijing (see the beginning of this preface) contributed substantially by bringing the researchers together. The work by the committees of CG 2008 was essential for this publication. Finally, the editors happily acknowledge the generous sponsors Beijing Longlife Group, Chinese Association for Artificial Intelligence, Northeastern University, Beijing University of Posts and Telecommunications, Universiteit Maastricht, ChessBase, ICGA, and IFIP WG 14.4 Games & Entertainment Computing.

July 2008

Jaap van den Herik
Xinhe Xu
Zongmin Ma
Mark Winands

Organization

Executive Committee

Editors	H. Jaap van den Herik Xinhe Xu Zongmin Ma Mark H.M. Winands
Program Co-chairs	H. Jaap van den Herik Xinhe Xu Zongmin Ma Mark H.M. Winands

Organizing Committee

Xinhe Xu (Chair)	Zhiqing Liu (Co-chair)
Johanna W. Hellemons	H. Jaap van den Herik
Mark H.M. Winands	

Sponsors

Main Sponsor	Beijing Longlife Group
Institutional Sponsors	Chinese Association for Artificial Intelligence Northeastern University Beijing University of Posts and Telecommunications Universiteit Maastricht ChessBase, Hamburg, Germany ICGA
Technical Sponsor	IFIP WG 14.4 Games & Entertainment Computing

Programme Committee

Yngvi Björnsson	Jeroen Donkers	Guy Haworth
Bruno Bouzy	Haw-ren Fang	Ryan Hayward
Ivan Bratko	Aviezri Fraenkel	Jaap van den Herik
Michael Buro	James Glenn	Shun-Chin Hsu
Tristan Cazenave	Pedro Gonzalez-Calero	Tsan-sheng Hsu
Keh-Hsun Chen	Michael Greenspan	Ming Huang
Paolo Ciancarini	Reijer Grimbergen	Hiroyuki Iida
Rémi Coulom	Tsuyoshi Hashimoto	Wijnand IJsselsteijn

Graham Kendall
Akihiro Kishimoto
Clyde Kruskal
Hans Kuijf
Jong Weon Lee
Yibo Li
Shun-Shii Lin
Zhiqing Liu
Ulf Lorenz
Zongmin Ma
Frans Morsch
Martin Müller
Anton Nijholt
Jacques Pitrat
Christian Posthoff
Matthias Rauterberg
Jonathan Schaeffer
Pieter Spronck
Nathan Sturtevant
Gerald Tesauro
Jos Uiterwijk
Mark Winands
I-Chen Wu
Xinhe Xu
Shi-Jim Yen
Jan van Zanten

Referees

Vadim Anshelevich
Ronald Bjarnason
Yngvi Björnsson
Marc Boule
Bruno Bouzy
Ivan Bratko
Michael Buro
Arthur Cater
Tristan Cazenave
Guillaume Chaslot
Keh-Hsun Chen
Paolo Ciancarini
Alessandro Cincotti
Rémi Coulom
Omid David-Tabibi
Arie de Bruin
Jeroen Donkers
Peter van Emde Boas
Gunnar Farnebäck
Haw-ren Fang
Aviezri Fraenkel
James Glenn
Pedro Gonzalez-Calero
Reijer Grimbergen
Matej Guid
Dap Hartmann
Tsuyoshi Hashimoto
Guy Haworth
Ryan Hayward
Philip Henderson
Shun-Chin Hsu
Tsan-sheng Hsu
Hiroyuki Iida
Graham Kendall
Akihiro Kishimoto
Levente Kocsis
Clyde Kruskal
Hans Kuijf
Jong Weon Lee
Robert Levinson
Alvin Levy
Łukasz Lew
Shun-Shii Lin
Zhiqing Liu
Richard Lorentz
Ulf Lorenz
Zongmin Ma
Stefan Meyer-Kahlen
Frans Morsch
Martin Müller
Xiaozhen Niu
Kohei Noshita
Gian-Carlo Pascutto
Wim Pijls
Jacques Pitrat
Christian Posthoff
Eric Postma
Jean-François Raskin
Matthias Rauterberg
Kees van Reeuwijk
Jeff Rollason
Aleksander Sadikov
Jahn-Takeshi Saito
Maarten Schadd
Jonathan Schaeffer
David Silver
Stephen Smith
Pieter Spronck
Renze Steenhuisen
Nathan Sturtevant
Pascal Tang
Gerald Tesauro
Jos Uiterwijk
Erik van der Werf
Jan Willemson
Thomas Wolf
I-Chen Wu
Xinhe Xu
Shi-Jim Yen

Table of Contents

Single-Player Monte-Carlo Tree Search

Maarten P.D. Schadd, Mark H.M. Winands, H. Jaap van den Herik,
Guillaume M.J.-B. Chaslot, and Jos W.H.M. Uiterwijk

Games and AI Group, MICC, Faculty of Humanities and Sciences,
Universiteit Maastricht, Maastricht, The Netherlands
{maarten.schadd,m.winands,herik,g.chaslot,uiterwijk}@micc.unimaas.nl

Abstract. Classical methods such as A* and IDA* are a popular and successful choice for one-player games. However, they fail without an accurate admissible evaluation function. In this paper we investigate whether Monte-Carlo Tree Search (MCTS) is an interesting alternative for one-player games where A* and IDA* methods do not perform well. Therefore, we propose a new MCTS variant, called Single-Player Monte-Carlo Tree Search (SP-MCTS). The selection and backpropagation strategy in SP-MCTS are different from standard MCTS. Moreover, SP-MCTS makes use of a straightforward Meta-Search extension. We tested the method on the puzzle SameGame. It turned out that our SP-MCTS program gained the highest score so far on the standardized test set.

1 Introduction

Recently, Monte-Carlo (MC) methods have become a popular approach for intelligent play in games. MC simulations have first been used as an evaluation function inside a classical search tree [4,5]. In this role, MC simulations have been applied to Backgammon [23], Clobber [18], and Phantom Go [6]. Due to the costly evaluation, the search is not able to investigate the search tree sufficiently deep in some games [4].

Therefore, the MC simulations have been placed into a tree-search context in multiple ways [9,10,17]. The resulting general method is called Monte-Carlo Tree Search (MCTS). It is a best-first search where the MC simulations guide the search. Especially in the game of Go, which has a large search space [3], MCTS methods are successful [9,10].

So far, MCTS has been applied rarely in one-player games. The only example we know of is the Sailing Domain [17]. There, it is applied on a game with uncertainty. So, to the best of our knowledge, MCTS has not been used in a one-player game with *perfect information* (a puzzle[1]). The traditional approaches to puzzles [16] are applying A* [14] or IDA* [19]. These methods have been quite successful for solving puzzles. The disadvantage of the methods is that they need an admissible heuristic evaluation function. The construction of such a function

[1] Although it is somewhat arbitrary, we will call these one-player games with perfect information for the sake of brevity *puzzles* [16].

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 1–12, 2008.

can be difficult. Since MCTS does not need an admissible heuristic, it may be an interesting alternative. In this paper we will investigate the application of MCTS to a puzzle. We introduce a new MCTS variant called SP-MCTS. The puzzle SameGame [20] will be used as a test domain in the remainder of this paper.

In Sect. 2 we present the background and rules of SameGame. In Sect. 3 we discuss why classical approaches are not suitable for SameGame. Then we introduce our SP-MCTS approach in Sect. 4. Experiments and results are given in Sect. 5. Section 6 shows our conclusions and indicates future research.

2 SameGame

Below, we first present some background information on SameGame, in Subsection 2.1. Subsequently we explain the rules in Subsection 2.2.

2.1 Background

SameGame is a puzzle invented by Kuniaki Moribe under the name *Chain Shot!* in 1985. It was distributed for Fujitsu FM-8/7 series in a monthly personal computer magazine called *Gekkan ASCII* [20]. The puzzle was afterwards recreated by Eiji Fukumoto under the name of *SameGame* in 1992. So far, the best program for SameGame has been developed by Billings [24].

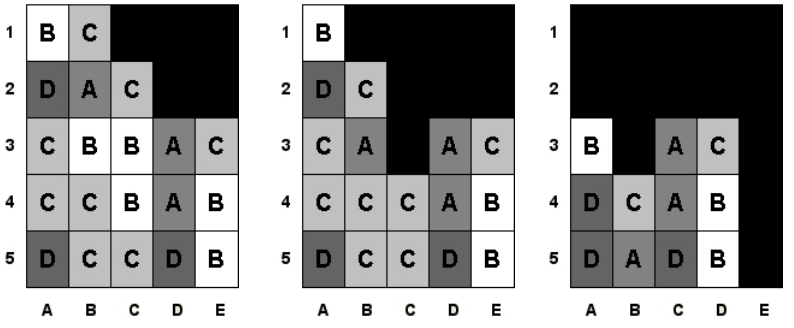
By randomly playing 10^6 puzzles, we estimated the average length of the game to be 64.4 moves and the average branching factor to be 20.7, resulting in a game-tree complexity of 10^{85}. Moreover, we computed the state-space complexity of the game to be 10^{159}.

2.2 Rules

SameGame is played on a rectangular vertically placed 15×15 board initially filled with blocks of 5 colors at random. A move consists of removing a group of (at least two) orthogonally adjacent blocks of the same color. The blocks on top of the removed group will fall down. As soon as an empty column occurs, the columns to the right of the empty column are shifted to the left. Therefore, it is impossible to create separate subgames. For each removed group points are rewarded. The amount of points is dependent on the number of blocks removed and can be computed by the formula $(n-2)^2$, where n is the size of the removed group.

We show two example moves in Fig. 1. When the 'B' group in the third column with a connection to the second column of position 1(a) is played, it will be removed from the game. In the second column the CA blocks will fall down and in the third column the 'C' block will fall down, resulting in position 1(b). Because of this move, it is now possible to remove a large group of 'C' blocks (n=6). Owing to an empty column the two columns at the right side of the board are shifted to the left, resulting in position 1(c).[2] The first move is worth 1 point; the second move is worth 16 points.

[2] Shifting the columns at the left side to the right would not have made a difference in points. For consistency, we will always shift columns to the left.



(a) Playing 'B' in the center column

(b) Playing 'C' in the center column

(c) Resulting position

Fig. 1. Example SameGame moves

The game is over if no more blocks can be removed. This happens when either the player (1) has removed all blocks or (2) is left with a position where no adjacent blocks have the same color. In the first case, 1,000 bonus points are rewarded. In the second case, points will be deducted. The formula for deducting is similar to the formula for awarding points but now iteratively applied for each color left on the board. Here it is assumed that all blocks of the same color are connected.

There are variations that differ in board size and the number of colors, but the 15×15 variant with 5 colors is the accepted standard. If a variant differs in scoring function, it is named differently (e.g., Jawbreaker, Clickomania) [1,21].

3 Classical Methods: A* and IDA*

The classical approach to puzzles involves techniques such as A* [14] and IDA* [19]. A* is a best-first search where all nodes have to be stored in a list. The list is sorted by an admissible evaluation function. At each iteration the first element is removed from the list and its children are added to the sorted list. This process is continued until the goal state arrives at the start of the list.

IDA* is an iterative deepening variant of A* search. It uses a depth-first approach in such a way that there is no need to store the complete tree in memory. The search will continue depth-first until the cost of arriving at a leaf node and the value of the evaluation function pass a certain threshold. When the search returns without a result, the threshold is increased.

Both methods are heavily dependent on the quality of the evaluation function. Even if the function is an admissible under-estimator, it still has to give an accurate estimation. Classical puzzles where this approach works well are the Eight Puzzle with its larger relatives [19,22] and Sokoban [15]. Here a good under-estimator is the well-known Manhattan Distance. The main task in this field of research is to improve the evaluation function, e.g., with pattern databases [11,12].

These classical methods fail for SameGame because it is not easy to make an admissible under-estimator that still gives an accurate estimation. An attempt

to make such an evaluation function is by just awarding points to the groups on the board without actually playing a move. However, if an optimal solution to a SameGame problem has to be found, we may argue that an "over-estimator" of the position is needed. An admissible "over-estimator" can be created by assuming that all blocks of the same color are connected and would be able to be removed at once. This function can be improved by checking whether there is a color with only one block remaining on the board. If this is the case, the 1,000 bonus points for clearing the board may be deducted because the board cannot be cleared completely. However, such an evaluation function is far from the real score on a position and does not give good results with A* and IDA*. Tests have shown that using A* and IDA* with the proposed "over-estimator" result in a kind of breadth-first search. The problem is that after expanding a node, the heuristic value of a child is significantly lower than the value of its parent, unless a move removes all blocks with one color from the board. We expect that other Depth-First Branch-and-Bound methods [25] suffer from the same problem.

Since no good evaluation function has been found yet, SameGame presents a new challenge for the puzzle research. In the next section we will discuss our SP-MCTS.

4 Monte-Carlo Tree Search

This section first gives a description of SP-MCTS in Subsection 4.1. Thereafter we will explain the Meta-Search extension in Subsection 4.2.

4.1 SP-MCTS

MCTS is a best-first search method, which does not require a positional evaluation function. MCTS builds a search tree employing Monte-Carlo evaluations at the leaf nodes. Each node in the tree represents an actual board position and typically stores the average score found in the corresponding subtree and the number of visits. MCTS constitutes a family of tree-search algorithms applicable to the domain of board games [9,10,17].

In general, MCTS consists of four steps, repeated until time has run out [7]. (1) A *selection strategy* is used for traversing the tree from the root to a leaf. (2) A *simulation strategy* is used to finish the game starting from the leaf node of the search tree. (3) The *expansion strategy* is used to determine how many and which children are stored as promising leaf nodes in the tree. (4) Finally, the result of the MC evaluation is propagated backwards to the root using a *back-propagation strategy*.

Based on MCTS, we propose an adapted version for puzzles: Single-Player Monte-Carlo Tree Search (SP-MCTS). Below, we will discuss the four corresponding phases and point out differences between SP-MCTS and MCTS.

Selection Strategy. Selection is the strategic task to select one of the children of a given node. It controls the balance between *exploitation* and *exploration*. Exploitation is the task to focus on the move that led to the best results so

far. Exploration deals with the less promising moves that still may have to be explored, due to the uncertainty of their evaluation so far. In MCTS at each node starting from the root, a child has to be selected until a leaf node is reached. Several algorithms have been designed for this setup [9,10].

Kocsis and Szepesvári [17] proposed the selection strategy UCT (Upper Confidence bounds applied to Trees). For SP-MCTS, we use a modified UCT version. At the selection of node N with children N_i, the strategy chooses the move, which maximizes the following formula.

$$\overline{X} \quad + \quad C \cdot \sqrt{\frac{ln\, t\,(N)}{t\,(N_i)}} \quad + \quad \sqrt{\frac{\sum x^2 - t\,(N_i) \cdot \overline{X}^2 + D}{t\,(N_i)}}. \tag{1}$$

The first two terms constitute the original UCT formula. It uses the number of times $t\,(N)$ that node N was visited and the number of times $t\,(N_i)$ that child N_i was visited to give an upper confidence bound for the average game value $\overline{X}$. For puzzles, we added a third term, which represents a possible deviation of the child node [8,10]. It contains the sum of the squared results so far $\left(\sum x^2\right)$ achieved in the child node corrected by the expected results $t\,(N_i) \cdot \overline{X}^2$. A high constant D is added to make sure that nodes, which have been rarely explored, are considered uncertain. Below we describe two differences between puzzles and two-player games, which may affect the selection strategy.

First, the essential difference between puzzles and two-player games is the *range of values*. In two-player games, the results of a game is denoted by *loss*, *draw*, or *win*, i.e., $\{-1,\ 0,\ 1\}$. The average score of a node will always stay within $[-1,1]$. In a puzzle, an arbitrary score can be achieved that is not by definition in a preset interval. In SameGame there are positions, which result in a value above 4,000 points. As a solution to this issue we may set the constants (C,D) in such a way that they are feasible for the interval $[0,\ 5000]$. A second solution would be to scale the values back into the above mentioned interval $[-1,1]$, given the maximum score of approximately 5,000 for a position. When the exact maximum score is not known a theoretical upper bound can be used. For instance, in SameGame a theoretical upper bound is to assume that all blocks have the same color. A direct consequence of such a high upperbound is that the game scores will be located near to zero. It means that the constants C and D have to be set with completely different values compared to two-player games. We have opted for the first solution in our program.

A second difference is that puzzles do not have any *uncertainty on the opponent's play*. It means that the line of play has to be optimized without the hindrance of an opponent.

Here we remark that Coulom [10] chooses a move according to the selection strategy only if $t\,(N)$ reaches a certain threshold (we set this threshold to 10). Before we cross the threshold, the simulation strategy is used. The latter is explained below.

Simulation Strategy. Starting from a leaf node, random moves are played until the end of the game. In order to improve the quality of the games, the moves are chosen pseudo-randomly based on heuristic knowledge.

In SameGame, we have designed two static simulation strategies. We named these strategies "TabuRandom" and "TabuColorRandom". Both strategies aim at making large groups of one color. In SameGame, making large groups of blocks is advantageous.

"TabuRandom" chooses a random color at the start of a simulation. It is not allowed to play this color during the random simulations unless there are no other moves possible. With this strategy large groups of the chosen color will be formed automatically.

The new aspect in the "TabuColorRandom" strategy with respect to the previous strategy is that the chosen color is the color most frequently occurring at the start of the simulation. This may increase the probability of having large groups during the random simulation.

Expansion Strategy. When a leaf node is reached, the expansion strategy decides on which nodes are stored in memory. Coulom [10] proposed to expand one child per simulation. With his strategy, the expanded node corresponds to the first encountered position that was not present in the tree. This is also the strategy we used for SameGame.

Back-Propagation Strategy. During the back-propagation phase, the result of the simulation at the leaf node is propagated backwards to the root. Several back-propagation strategies have been proposed in the literature [9,10]. The best results that we have obtained was by using the plain average of the simulations. Therefore, we update (1) the average score of a node. Additional to this, we also update (2) the sum of the squared results because of the third term in the selection strategy (see Formula 1), and (3) the best score achieved so far for computational reasons.

The four phases are iterated until the time runs out.[3] When this happens, a final move selection is used to determine which move should be played. In two-player games (with an analogous run-out-of-time procedure) the best move according to this strategy will be played by the player to move and the opponent then has time to calculate his response. But in puzzles this can be done differently. In puzzles it is not needed to wait for an unknown reply of an opponent. Because of this, it is possible to perform one large search from the initial position and then play all moves at once. With this approach all moves at the start are under consideration until the time for SP-MCTS runs out.

4.2 Meta-search

A Meta-Search is a search method that does not perform a search on its own but uses other search processes to arrive at an answer. For instance, Gomes *et al.*

[3] In general, there is no time limitation for puzzles. However, a time limit is necessary to make testing possible.

[13] proposed a form of iterative deepening to handle heavy-tailed scheduling tasks. The problem was that the search was lost in a large subtree, which would take a large amount of time to perform, while there are shallow answers in other parts of the tree. The possibility exists that by restarting the search a different part of the tree was searched with an easy answer.

We discovered that it is important to generate deep trees in SameGame (see Subsection 5.2). However, by exploiting the most-promising lines of play, the SP-MCTS can be caught in local maxima. So, we extended SP-MCTS with a straightforward form of Meta-Search to overcome this problem. After a certain amount of time, SP-MCTS just restarts the search with a different random seed. The best path returned at the end of the Meta-Search is the path with the highest score found in the searches. Subsection 5.3 shows that this form of Meta-Search is able to increase the average score significantly.

5 Experiments and Results

Subsection 5.1 shows tests of the quality of the two simulation strategies TabuRandom and TabuColorRandom. Thereafter, the results of the parameter tuning are presented in Subsection 5.2. Next, in Subsection 5.3 the performance of the Meta-Search on a set of 250 positions is shown. Finally, Subsection 5.4 compares SP-MCTS to IDA* and Depth-Budgeted Search (used in the program by Billings [2]).

5.1 Simulation Strategy

In order to test the effectiveness of the two simulation strategies we used a test set of 250 randomly generated positions.[4] We applied SP-MCTS without the Meta-Search extension for each position until 10 million nodes were reached in memory. These runs typically take 5 to 6 minutes per position. The best score found during the search is the final score for the position. The constants C and D were set to 0.5 and 10,000, respectively. The results are shown in Table 1.

Table 1 shows that the TabuRandom strategy has a significant better average score (i.e., 700 points) than plain random. Using the TabuColorRandom strategy the average score is increased by another 300 points. We observe that a low standard deviation is achieved for the random strategy. In this case, it implies that all positions score almost equally low.

Table 1. Effectiveness of the simulation strategies

	Random	TabuRandom	TabuColorRandom
Average Score	2,069	2,737	3,038
StdDev	322	445	479

[4] The test set can be found at www.cs.unimaas.nl/maarten.schadd/TestSet.txt

5.2 SP-MCTS Parameter Tuning

This subsection presents the parameter tuning in SP-MCTS. Three different settings were used for the pair of constants $(C; D)$ of Formula 1, in order to investigate which balance between exploitation and exploration gives the best results. These constants were tested with three different time controls on the test set of 250 positions, expressed by a maximum number of nodes. The three numbers are 10^5, 10^6 and 5×10^6. The short time control refers to a run with a maximum of 10^5 nodes in memory. In the medium time control, 10^6 nodes are allowed in memory, and in long time control 5×10^6 nodes are allowed. We have chosen to use nodes in memory as measurement to keep the results hardware-independent. The parameter pair (0.1; 32) represents *exploitation*, (1; 20,000) performs *exploration*, and (0.5; 10,000) is a balance between the other two.

Table 2 shows the performance of the SP-MCTS approach for the three time controls. The short time control corresponds to approximately 20 seconds per position. The best results are achieved by exploitation. The score is 2,552. With this setting the search is able to build trees that have on average the deepest leaf node at ply 63, implying that a substantial part of the chosen line of play is inside the SP-MCTS tree. Also, we see that the other two settings are not generating a deep tree.

In the medium time control, the best results were achieved by using the balanced setting. It scores 2,858 points. Moreover, Table 2 shows that the average score of the balanced setting increased most compared to the short time control, viz. 470. The balanced setting is now able to build substantially deeper trees than in the short time control (37 vs. 19). An interesting observation can be made by comparing the score of the exploration setting in the medium time

Table 2. Results of SP-MCTS for different settings

10^5 nodes	Exploitation (0.1; 32)	Balanced (0.5; 10,000)	Exploration (1; 20,000)
Average Score	2,552	2,388	2,197
Standard Deviation	572	501	450
Average Depth	25	7	3
Average Deepest Node	63	19	8
10^6 nodes	(0.1; 32)	(0.5; 10,000)	(1; 20,000)
Average Score	2,674	2,858	2,579
Standard Deviation	607	560	492
Average Depth	36	14	6
Average Deepest Node	71	37	15
5×10^6 nodes	(0.1; 32)	(0.5; 10,000)	(1; 20,000)
Average Score	2,806	3,008	2,901
Standard Deviation	576	524	518
Average Depth	40	18	9
Average Deepest Node	69	59	20

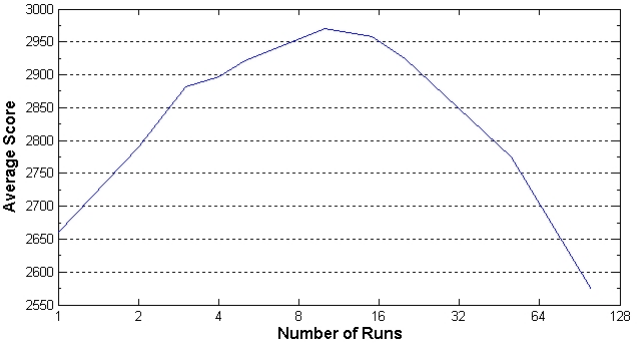


Fig. 2. The average score for different settings of the Meta-Search

control to the exploitation score in the short time control. Even with 10 times the amount of time, exploring is not able to achieve a significantly higher score than exploiting.

The results for the long experiment are that the balanced setting again achieves the highest score with 3,008 points. Now its deepest node on average is at ply 59. However, the exploitation setting only scores 200 points fewer than the balanced setting and 100 fewer than exploration.

From the results presented we may draw two conclusions. First we may conclude that it is important to have a deep search tree. Second, exploiting local maxima can be more advantageous than searching for the global maxima when the search only has a small amount of time.

5.3 Meta-search

This section presents the performance tests of the Meta-Search extension of SP-MCTS on the set of 250 positions. We remark that the experiments are time constrained. Each experiment could only use 5×10^5 nodes in total and the Meta-Search distributed these nodes fairly among the number of runs. It means that a single run can take all 5×10^5 nodes, but that two runs would only use 2.5×10^5 nodes each. We used the exploitation setting (0.1; 32) for this experiment. The results are depicted in Fig. 2.

Figure 2 indicates that already with two runs instead of one, a significant performance increase of 140 points is achieved. Furthermore, the maximum average score of the Meta-Search is at ten runs, which uses 5×10^4 nodes for each run. Here, the average score is 2,970 points. This result is almost as good as the best score found in Table 2, but with the difference that the Meta-Search used one tenth of the number of nodes. After ten runs the performance decreases because the generated trees are not deep enough.

5.4 Comparison

The best SameGame program so far has been written by Billings [2]. This program performs a non-documented method called Depth-Budgeted Search (DBS).

Table 3. Comparing the scores on the standardized test set

Position no.	IDA*	DBS	SP-MCTS	Position no.	IDA*	DBS	SP-MCTS
1	548	2,061	2,557	11	1,073	2,911	3,047
2	1,042	3,513	3,749	12	602	2,979	3,131
3	841	3,151	3,085	13	667	3,209	3,097
4	1,355	3,653	3,641	14	749	2,685	2,859
5	1,012	3,093	3,653	15	745	3,259	3,183
6	843	4,101	3,971	16	1,647	4,765	4,879
7	1,250	2,507	2,797	17	1,284	4,447	4,609
8	1,246	3,819	3,715	18	2,586	5,099	4,853
9	1,887	4,649	4,603	19	1,437	4,865	4,503
10	668	3,199	3,213	20	872	4,851	4,853
				Total:	22,354	72,816	73,998

When the search reaches a depth where its budget has been spent, a greedy simulation is performed. On a standardized test set of 20 positions[5] his program achieved a total score of 72,816 points with 2 to 3 hours computing time per position. Using the same time control, we tested SP-MCTS on this set. We used again the exploitation setting (0.1; 32) and the Meta-Search extension, which applied 1,000 runs using 100,000 nodes for each search process. For assessment, we tested IDA* using the evaluation function described in Sect. 3. Table 3 compares IDA*, DBS, and SP-MCTS with each other.

SP-MCTS outperformed DBS on 11 of the 20 positions and was able to achieve a total score of 73,998. Furthermore, Table 3 shows that IDA* does not perform well on this puzzle. It plays at human beginner level. The best variants discovered by SP-MCTS can be found on our website.[6] There we see that SP-MCTS is able to clear the board for all 20 positions. This confirms that a deep search tree is important for SameGame as was seen in Subsection 5.2.

By combining the scores of DBS and SP-MCTS we computed that at least 75,152 points can be achieved for this set.

6 Conclusions and Future Research

In this paper we proposed a new MCTS variant called Single-Player Monte-Carlo Tree Search (SP-MCTS). We adapted MCTS by two modifications resulting in SP-MCTS. The modifications are (1) the selection strategy and (2) the back-propagation strategy. Below we provide three observations and subsequently two conclusions.

First, we observed that our TabuColorRandom strategy significantly increased the score of the random simulations in SameGame. Compared to the pure random simulations, an increase of 50% in the average score is achieved.

[5] The positions can be found at www.js-games.de/eng/games/samegame

[6] The best variations can be found at http://www.cs.unimaas.nl/maarten.schadd/SameGame/Solutions.html

Next, we observed that it is important to build deep SP-MCTS trees. Exploiting works better than exploring at short time controls. At longer time controls the balanced setting achieves the highest score, and the exploration setting works better than the exploitation setting. However, exploiting the local maxima still leads to comparable high scores.

Third, with respect to the extended SP-MCTS endowed with a straightforward Meta-Search, we observed that for SameGame combining a large number of small searches can be more beneficial than doing one large search.

From the results of SP-MCTS with parameters (0.1; 32) and with Meta-Search set on a time control of around 2 hours we may conclude that SP-MCTS produced the highest score found so far for the standardized test set. It was able to achieve 73,998 points, meanwhile breaking Billings' record by 1,182 points.

A second conclusion is that we have shown that SP-MCTS is applicable to a one-person perfect-information game. SP-MCTS is able to achieve good results in SameGame. So, SP-MCTS is a worthy alternative for puzzles where a good admissible estimator cannot be found.

In the future, more techniques will be tested on SameGame. We mention three of them. First, knowledge can be included in the selection mechanism. A technique to achieve this is called *Progressive Unpruning* [7]. Second, this paper demonstrated that combining small searches can achieve better scores than one large search. However, there is no information shared between the searches. This can be achieved by using a transposition table, which is not cleared at the end of a small search. Third, the Meta-Search can be parallelized asynchronously to take advantage of multi-processor architectures.

Acknowledgments. This work is funded by the Dutch Organisation for Scientific Research (NWO) in the framework of the project TACTICS, grant number 612.000.525.

References

1. Biedl, T.C., Demaine, E.D., Demaine, M.L., Fleischer, R., Jacobsen, L., Munro, I.: The Complexity of Clickomania. In: Nowakowski, R.J. (ed.) More Games of No Chance, Proc. MSRI Workshop on Combinatorial Games, pp. 389–404. MSRI Publ., Berkeley. Cambridge University Press, Cambridge (2002)
2. Billings, D.: Personal Communication. University of Alberta, Canada (2007)
3. Bouzy, B., Cazanave, T.: Computer Go: An AI-Oriented Survey. Artificial Intelligence 132(1), 39–103 (2001)
4. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Proceedings of the 10th Advances in Computer Games Conference (ACG-10), The Netherlands, pp. 159–174. Kluwer Academic, Dordrecht (2003)
5. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)
6. Cazenave, T., Borsboom, J.: Golois Wins Phantom Go Tournament. ICGA Journal 30(3), 165–166 (2007)

7. Chaslot, G.M.J.-B., Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J., Bouzy, B.: Progressive strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation 4(3), 343–357 (2008)
8. Chaslot, G.M.J.B., de Jong, S., Saito, J.-T., Uiterwijk, J.W.H.M.: Monte-Carlo Tree Search in Production Management Problems. In: Schobbens, P.Y., Vanhoof, W., Schwanen, G. (eds.) Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium, pp. 91–98 (2006)
9. Chaslot, G.M.J.B., Saito, J.-T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo Strategies for Computer Go. In: Schobbens, P.Y., Vanhoof, W., Schwanen, G. (eds.) Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium, pp. 83–91 (2006)
10. 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)
11. Culberson, J.C., Schaeffer, J.: Pattern databases. Computational Intelligence 14(3), 318–334 (1998)
12. Felner, A., Zahavi, U., Schaeffer, J., Holte, R.C.: Dual Lookups in Pattern Databases. In: IJCAI, Edinburgh, Scotland, pp. 103–108 (2005)
13. Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. In: AIPS, Pittsburg, PA, pp. 208–213 (1998)
14. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernatics 4(2), 100–107 (1968)
15. Junghanns, A.: Pushing the Limits: New Developments in Single Agent Search. PhD thesis, University of Alberta, Alberta, Canada (1999)
16. Kendall, G., Parkes, A., Spoerer, K.: A Survey of NP-Complete Puzzles. ICGA Journal 31(1), 13–34 (2008)
17. 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)
18. Kocsis, L., Szepesvári, C., Willemson, J.: Improved Monte-Carlo Search (2006), http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf
19. Korf, R.E.: Depth-first iterative deepening: An optimal admissable tree search. Artificial Intelligence 27(1), 97–109 (1985)
20. Moribe, K.: Chain shot! Gekkan ASCII, (November 1985) (in Japanese)
21. PDA Game Guide.com. Pocket PC Jawbreaker Game. The Ultimate Guide to PDA Games, Retrieved 7.1.2008 (2008), http://www.pdagameguide.com/jawbreaker-game.html
22. Sadikov, A., Bratko, I.: Solving 20 × 20 Puzzles. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computer Games Workshop 2007 (CGW 2007), The Netherlands, pp. 157–164. Universiteit Maastricht, Maastricht (2007)
23. Tesauro, G., Galperin, G.R.: On-line policy improvement using Monte Carlo search. In: Mozer, M.C., Jordan, M.I., Petsche, T. (eds.) Advances in Neural Information Processing Systems, vol. 9, pp. 1068–1074. MIT Press, Cambridge (1997)
24. University of Alberta GAMES Group. GAMES Group News (Archives) (2002), http://www.cs.ualberta.ca/~games/archives.html
25. Vempaty, N.R., Kumar, V., Korf, R.E.: Depth-first versus best-first search. In: AAAI, Anaheim, California, USA, pp. 434–440. MIT Press, Cambridge (1991)

Amazons Discover Monte–Carlo

Richard J. Lorentz

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

Abstract. Monte-Carlo algorithms and their UCT-like successors have recently shown remarkable promise for Go-playing programs. We apply some of these same algorithms to an Amazons-playing program. Our experiments suggest that a pure MC/UCT type program for playing Amazons has little promise, but by using strong evaluation functions we are able to create a hybrid MC/UCT program that is superior to both the basic MC/UCT program and the conventional minimax-based programs. The MC/UCT program is able to beat INVADER, a strong minimax program, over 80% of the time at tournament time controls.

1 Introduction

Amazons is a fairly new game, invented in 1988 by Walter Zamkauskas of Argentina and is a trademark of Ediciones de Mente. Amazons has attracted attention in the game-programming community because of its simple rules and its difficulty of play. On average there are more than 1,000 legal moves at any given position. Typically many hundreds of these legal moves seem quite reasonable [1]. A game lasts a maximum of 92 moves but is usually decided by about move 40, so games can be completed in a reasonable amount of time. Most people view Amazons as intermediate in difficulty between Chess and Go. Thus, Amazons provides a natural test bed for games research.

To date, most strong Amazons programs use traditional minimax-based algorithms. Two such programs are INVADER [1] and AMAZONG [12, 13]. However, Monte-Carlo (MC)-based algorithms have recently been used extremely effectively in Go programming [5, 9]. In fact, MC-based programs now completely dominate in the arena of 9×9 Go, and are beginning to do the same in 19×19 Go. It seems natural to test similar techniques in Amazons programming. Kloetzer, Iida, and Bouzy were the first to try this with their Amazons program CAMPYA [10]. We have modified INVADER, our Amazons playing program, to create INVADERMC, an MC-based Amazons program.

In the following we discuss the details of our approach. We explain how MC was added to INVADER and describe improvements to the basic MC algorithm that proved useful to INVADERMC. These include forward pruning and stopping MC simulations early. We then explain how UCT is added and discuss improvements that include finding the proper evaluation function, progressive widening, choosing the correct moment to stop a random simulation, and other tuning issues. We show the level of success we have achieved, and how the techniques we used in INVADERMC compare to those used in CAMPYA.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 13–24, 2008.

In the next section we review the rules of Amazons. We then summarize the main features of MC algorithms and briefly describe the UCT enhancement. Our emphasis throughout is on how these algorithms apply to Amazons programming. Section 3 provides all the details of our approach. We focus on the techniques that proved particularly effective for INVADERMC. We conclude in Sect. 4 with a summary of what we accomplished, a description of techniques that surprised us as not being particularly effective, and suggestions for further research.

2 The Game of Amazons and the Monte-Carlo Approach

This section provides basic background information. We explain the rules of Amazons and briefly describe the MC and UCT algorithms.

2.1 The Rules of Amazons

Amazons is usually played on a 10×10 board where each player is given 4 amazons that are initially placed as shown in Fig. 1(a). Although the game may be played with other board sizes and with different numbers and different initial placements of amazons, this version is considered standard and so will be the only version discussed here.

A move comprises two steps: first an amazon is moved like a chess queen through unoccupied squares to an unoccupied square. Then from this new location the amazon "throws an arrow", again like a chess queen, across unoccupied squares to an unoccupied square where it stays for the remainder of the game. The arrow now occupies that square, acting as a barrier through which neither amazons nor thrown arrows can pass. From the initial position shown in Fig. 1(a), one of White's 2,176 possible moves is to move the amazon from square D1 to square D8 and then throw an arrow to square B6, where the arrow must stay for the remainder of the game. This move is denoted D1-D8(B6) and is shown in Fig. 1(b). White moves first and passing is not allowed. The last player to move wins.

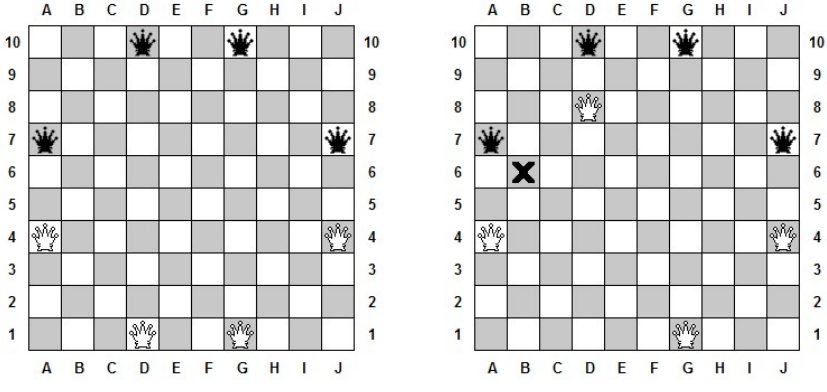


Fig. 1. The initial position and a typical first move

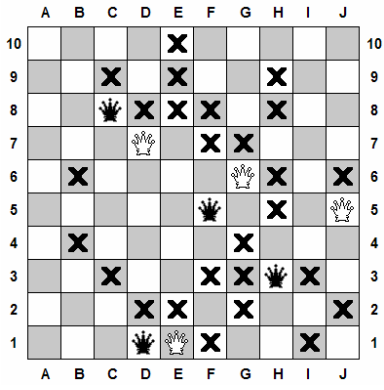


Fig. 2. MC algorithm failure

The concept of territory is important in Amazons. Territory concerns regions of the board that are under the complete control of one player, thus allowing that player to be the only player to be moving amazons and throwing arrows there. For example, in Fig. 2 below, if White moves J5-I5(J5) White will have complete control of the upper right region and this region will be considered White's territory. White can make 16 undisturbed moves there. The upper left side of the board in Fig. 2 appears likely to become eventually Black's territory and so we might say that Black has approximately seven squares of potential territory there. Obviously acquiring territory is an important aspect of the game since the side with the most territory should be able to make the last move and win the game.

2.2 Monte-Carlo Programming

The MC approach to game programming is well known and has been used in Go, for example, with varying degrees of success for more than 10 years [2]. The basic idea is that the best move is found by repeatedly sampling all possible moves either iteratively, random uniformly, or more cleverly. For each move sampled a random game is played to the end. This random game is sometimes referred to as a simulation. The wins and losses of the various simulations are tabulated and eventually the move that reports the highest winning percentage is played. In Amazons, as with Go, this simple algorithm can be surprisingly effective. If more random games are played, more information is acquired, so the speed of playing out random games is significant.

It is important to understand that a pure MC approach has a fatal flaw as can be seen in a game that was played by our first version of INVADERMC. The critical position is shown in Fig. 2 where INVADERMC is White.

From the position in Fig. 2, White made the move G6-C6(G6). This loses immediately because Black can now play H3-I4(I6) closing off White's potential territory in the upper right. But that is the only move that wins for Black, so White still wins the vast majority of the random games after making that faulty move. If instead White plays a move like J5-I5(J5), White will be winning, but the game is actually quite close. So White wins fewer of the random games and this winning move actually appears less desirable to INVADERMC.

2.3 The UCT Algorithm

One popular solution to this problem is referred to as UCT [11] and stands for "upper confidence bounds extended for trees". The idea is that instead of just keeping a list of candidate moves and running random games for each of them, we maintain a search tree where each node of the tree keeps track of the winning record for that particular position. We call this a UCT tree. Starting at the root of the UCT tree a path is found to a leaf node by proceeding down the tree, choosing nodes that have the highest expansion value. The method for calculating the expansion value is described below. A random simulation is then run from the position corresponding to this leaf node and this leaf may then be expanded, adding its children to the UCT tree. The decision to expand a leaf node is usually based on how many times that node was visited.

The expansion value of a node is equal to the winning percentage of all simulations that have passed through that node plus a bias value. The bias value allows nodes that have not been visited very often to obtain a share of the simulations and is calculated using the formula

$$k \cdot \sqrt{\frac{\log(parent\_count)}{node\_count}} \tag{1}$$

In this formula *parent_count* corresponds to the number of simulations that have passed through the parent of the node, *node_count* is the number of simulations that have passed through the node, and k is a constant that is tuned according to the particular application. A small value for k means moves that have been performing well will continue to be expanded with high likelihood (exploitation) while a large value means more moves tend to be tried at each level (exploration) [8].

For example, when using UCT from the position in Fig. 2, the node in the tree corresponding to the move G6-C6(G6) will have child nodes corresponding to each of Black's follow-ups. Since the child node corresponding to the move H3-I4(I6) will win most of its simulations, that path down the tree will be the one taken most often, meaning that the top node G6-C6(G6) will start losing more games, eventually showing it to be undesirable. Meanwhile, because of the bias value, another move like J5-I5(J5), will eventually be explored frequently to obtain a higher winning percentage and UCT will continue to grow the tree and confirm that that move (or something similar) is the better move in this position.

In the next section we explain in detail how we incorporated the basic ideas of MC/UCT into INVADERMC and then made improvements to create a hybrid MC/UCT program. This hybrid version is considerably stronger than minimax-based INVADER.

3 The Hybrid MC/UCT Approach

INVADER, our strong minimax-based Amazons program, has competed in numerous Computer Olympiads and has finished second or third every time. We used INVADER as the basis for creating INVADERMC, our MC/UCT based program. We will summarize the various basic algorithms and modifications to these algorithms we used and how they affected playing quality to show how we ultimately created a strong MC/UCT-based hybrid program. Unless otherwise stated all tests were 100-game

matches where each game of the match began with a different two-move opening. Games were played at tournament speeds, i.e., at the rate of 30 minutes for the entire game per player.

3.1 Adding MC to INVADER

Our experiments began with a pure MC-based version of INVADER. The idea was to see how strong our program could become by using only the MC-based idea, that is, by not building any trees as is required by more sophisticated techniques such as UCT. The most basic version simply repeatedly iterated over all legal moves, ran a random game from each such position, and kept track of the results. The main obstacle that needed to be overcome was to find a way to run random games quickly. There are usually over 1,000 legal moves in a typical Amazons position. So, finding a random move without bias among all the possibilities can be quite time consuming. Since an amazon's move is comprised of two parts, the move and the throw, we first select a random move uniformly among all the possible amazon moves and then from that move select a random throw. This is the same technique used in CAMPYA [10]. It does add a slight bias towards amazon moves that have fewer throws available, i.e., amazons that are in more restricted positions. But the cost of generating all possible legal moves slowed the random generation by more than a factor of 10 and so was deemed too slow for any further study.

There is an obvious way to speed this up even more. We can first randomly select one of the four amazons, then randomly select one of its moves, and finally select a throw from there. Moves produced using this method were of conspicuously lower quality and almost no test games were won using this approach. So, we immediately rejected this approach. All further tests were done using the two-step move generation process. This basic MC program did surprisingly well, winning 15% of the games in a test match against our min-max-based INVADER.

With MC-based programs the more simulations we can generate the more information we can gain. So, it is beneficial to be as efficient as possible. Subsections 3.2 and 3.5 deal with some other efficiency-related topics. However, we refrain from any discussion or tests concerning low-level efficiency concerns such as bit string board representations.

3.2 Stopping the MC Simulation Early

The basic MC program did quite well, but there are a number of things that can be done to improve the quality of its moves. The first has to do with choosing the correct time to end a simulation. Instead of waiting until a player loses because it has no legal moves we can stop a game as soon as all amazons have been completely isolated from its enemies. This gives a more precise and earlier determination but comes at a price of forcing us to check for this property during the simulations, thus slowing them down. It improved performance but suggested an even more important improvement.

The point of a random simulation is to provide statistical evidence about the strength of a move. Instead of simulating to the end of the game we can invoke an evaluation function earlier to help us determine whether the simulation is leading to a winning or a losing position. INVADER has a fairly strong evaluation function. So, it is

easy to test this premise by using it in INVADERMC. Note that this approach is never used in MC Go programming because evaluation functions are notoriously poor in Go. In Go the random simulations to the end of the game provide better information than an evaluation function.

When using the evaluation function we have a choice of recording either the presumed margin of victory, or simply who won. We choose the latter. In other words, only the sign of the evaluation is actually used at the end of the simulation. Though CAMPYA programmers claim to find advantage in using the margin of victory [10], so far we have been unable to improve on using the simple win/loss results.

But when is the correct time to invoke the evaluation function? The two obvious choices are: (1) after a certain stage of the game (e.g., after move number 30) has been reached; or (2) after a certain number of random moves have been played. Early experiments indicated that choice (2) produces a better INVADERMC, winning more than twice as many games against INVADER than various versions trying the other approach. Applying the evaluation after 6 random moves appears best. Waiting until 10 moves caused INVADERMC to win 10% fewer games and waiting until 14 moves dropped the win rate about 30%. Also, applying the evaluation after only 4 moves had a similar negative effect, dropping the win rate 25%. CAMPYA [10] does something similar, though it is not clear exactly how deep into the simulation they invoke their evaluation function.

3.3 Forward Pruning with MC

A second possible use for the evaluation function is to do (unsafe) forward pruning. Rather than waste time and space building the UCT tree for all legal moves we can use the evaluation to select the moves that are most likely to be good. With some tuning we found that selecting the top 400 moves for MC simulation seemed to give good results. More detailed tuning in the UCT case also found 400 to be a good choice. For example, our optimal UCT program that keeps the top 400 moves wins 80% of its games against normal INVADER. Instead, keeping the top 800 moves produces a slightly lower win rate, but when keeping 1,200 moves the win rate drops to 57%. Also, keeping only the top 200 moves shows a similar drop, winning only 61% of its games.

Extending this forward pruning idea, even among these 400 moves we select, we prefer gaining information about moves that are likely to be good and only occasionally simulate moves that appear to be weak, giving them a chance to prove themselves if necessary. We implemented this technique by essentially doing a depth-1 UCT search. We used the UCT bias values to determine which move we would simulate, but we never extended the search tree beyond its initial depth of 1.

Using these algorithms INVADERMC played at nearly the level of INVADER, winning 40% of its games. However, as pointed out in Sect. 2, there is a limit to how well we can expect a pure MC program to play.

3.4 Adding UCT

UCT extends the MC algorithm by creating a search tree of moves so that nodes (moves) that are performing well create child nodes beneath them and then the more

promising children are given more simulations. Where MC tends to concentrate on promising moves (nodes), UCT concentrates on promising variations (tree paths). The basic algorithms for UCT are straightforward but a great deal of care is needed to make the best use of UCT.

As mentioned in Subsection 2.3, it is important to choose the proper exploitation/exploration constant k. Experimental results indicate that in the case of INVADERMC, choosing the constant so that the UCT tree created ultimately ends up with a maximum depth of around 8 (assuming normal tournament time controls of 30 minutes per game) seems optimal. Setting $k = .35$ achieves this kind of result.

Attempts at deeper exploitation miss too many important refutations while more exploration does not give sufficient time for the best moves to reveal themselves. We compared various choices of k using our best INVADERMC which wins 80% of its games against INVADER. If we divide k by 2 the UCT tree now grows to depths of around 10 or 11, but INVADERMC only wins 63% of its games. If we multiply k by 2 the UCT depths tend to be around 6 but we obtain results very similar to those using the original k. This version won 78% of its games. However, if we multiply k by 4 we obtain a drastic change. This version has UCT tree depths of around 4 or 5 and only wins 25% of its games against INVADER.

By way of comparison, minimax-based INVADER is usually not able to play beyond depth 3 in the early stages of the game and can sometimes reach depth 8 or 9, but only at the latest stages. Minimax INVADER is, of course, implemented with most of the usual improvements including alpha-beta pruning, the killer heuristic, etc. It also prunes all but the best 400 moves at each node of the minimax tree. Since it is not clear what the relationship is between UCT depths and minimax depths this comparison is presented simply as a point of interest.

Also mentioned in Subsection 2.3, we need to decide when to expand leaf nodes in the tree. The idea to not necessarily expand a leaf the first time it is visited was first mentioned by Coulom in a slightly different context and is frequently used in MC/UCT Go programming [5]. There are a number of reasons why it is not prudent to expand immediately in the case of Amazons. First, to expand a node takes a great deal of time since all possible moves need to be generated and all of these moves need to then be evaluated. Experiments and intuition suggest this time might be better spent doing more simulations. A second reason is that even though we employ forward pruning and expand leaf nodes to have only 400 children, 400 is sufficiently large to cause the UCT tree to grow quite fast and can quickly exhaust available memory. Third, a single random Amazons simulation does not necessarily provide that much information about the likelihood that a given node is favorable for White or Black. We need to gather more data. Again, experiments with our best version of INVADERMC indicate that a leaf should not be expanded until it has been visited around 40 times. Dropping that value to 20 reduces the win rate by a small amount. Dropping it to 5 produces a program whose win rate is only 65% compared with 80% for our best INVADERMC. Raising the value to 120 has a similar negative effect.

By adding these basic UCT features to INVADERMC we have created a program that outperforms INVADER. This version of INVADERMC is able to beat INVADER approximately 60% of the time. We now describe further enhancements that move us closer to our best version that achieves a win rate of 80% against INVADER.

3.5 Choosing the Proper Evaluation Function

As is the case with most game-playing algorithms, the speed of the evaluation function is critical. Not only does an evaluation take place with every simulation, but every time a UCT tree node is expanded there will typically be on the order of 1,000 evaluations to perform corresponding to the average number of legal moves from a position. All of these nodes need evaluating so that we can select the top 400. There is an obvious tradeoff here: the faster the evaluation, the less information it provides so the more simulations we will need. But a faster evaluation also allows more simulations in the same amount of time. Our experience shows that very fast yet naïve evaluations allow significantly more simulations, but the lost information makes for a poorer player. Nevertheless, we were able to achieve some success by creating less precise but faster evaluation functions.

INVADER uses some quite complicated calculations to estimate accurately the territories of the players. One of the most time-consuming aspects involves doing a flood fill type calculation in each area of potential territory. We then rate the value of this potential territory based on factors such as the calculated size and the number of external access points to this potential territory.

By being less careful about the exact sizes and likelihoods of the players acquiring territory, but still taking territory into account on a rougher scale, we are able to evaluate a board in about two thirds of the time the normal evaluation takes. We do this by simply saying that a group of contiguous squares constitutes potential territory if every square in the group requires fewer queen moves to reach a friendly amazon than it takes to reach an enemy one. With this added speed INVADERMC is able to create a larger UCT tree while still evaluating reasonably accurately. INVADERMC using this faster evaluation function improves considerably, winning 67% of its test games against INVADER. It is worth noting that using the new evaluation function in the normal INVADER program makes INVADER weaker. It only won 30% of the games in a test against INVADER with its original evaluation. Also, our best INVADERMC program wins 89% of its games against INVADER using the new function. So the new evaluation benefits INVADERMC but harms INVADER.

3.6 Evaluation Parity Effect

Our simulations make a fixed number of random moves, usually 6, before the evaluation function is invoked. Since these simulations can commence from any leaf in the UCT tree, depending where we are in the tree, the simulation might terminate with either White or Black to play. This parity problem is particularly problematic with INVADERMC because its evaluation values swing quite widely according to whose turn it is to move. This ultimately causes the values that propagate up the UCT tree to be unnecessarily undependable. A useful fix is to terminate all random simulations with the same player to move. This means that, for example, when we are doing 6-move simulations we actually do 6 and 5-move simulations, depending on where in the tree we are relative to the starting position. This enhancement produces a significant improvement in the playing strength of INVADERMC. Our best INVADERMC without this enhancement only wins 73% of its games.

3.7 Progressive Widening

The final important enhancement has many names. We choose the term "progressive widening." The basic idea is to give UCT a little assistance by first considering only the "best" children of a node and then gradually adding the other children until at some point all children of a node are actually being examined by the UCT algorithm [3, 4, 6]. There are many ways to implement this, some quite sophisticated. We have implemented a very basic version. We see how many times a node has been visited and if this number is small only some of the children will be examined. As the node is visited more often, we gradually add more child nodes to be considered by UCT until we eventually end up considering all 400 children when the node has been visited sufficiently often. To give a sense of the actual values, on our machine INVADERMC usually constructs a UCT tree of about 1,500 nodes under normal time controls. Until a node has been visited 1,000 times (recall that leaves do not expand until they have been visited 40 times) we only consider 5 children. We increase the number of children by increments of 5 for every 1,000 visits and all 400 nodes are not considered until the node has been visited 80,000 times. Without this improvement our best INVADERMC wins fewer than 70% of its games against INVADER.

3.8 Minor Tuning

There are two other minor tuning issues that are interrelated and deserve mention. Though individually they do not contribute significantly to the strength of the program, combined they do seem to have a small positive effect. In most games, including Amazons, it is important to get off to a good start early. Therefore, many game programs, especially MC based programs, allocate more time for moves in the early stages of the game. INVADERMC does this in a primitive, but still useful way. Given the time control, we calculate how much time is available per move and then allow triple this base time for the first 10 moves, double for the next 10 moves, 1.5 times the base time for the next 10, the base time for the next 10 moves, and then allocate the remaining time to the remaining moves. However, the time allocated to these final moves is so small that often the UCT algorithm does not have time to build a sufficiently big tree to find a decent move. So, for these last moves we switch over to our basic minimax engine.

INVADERMC configured with all of the above features is significantly stronger than basic INVADER. After playing 400 games between the two, INVADERMC wins about 80% of the games.

4 Summary of Our Findings and Future Work

We have found that by adding a hybrid UCT structure to our Amazons playing program, INVADER, the new program, INVADERMC, is significantly stronger and is able to beat INVADER 80% of the time. It is not a pure UCT approach, because we prune (in an unsafe manner) child nodes and we run the random games for only about 6 moves. But our use of progressive widening and our tuning of the UCT depth and node visits per leaf expansion variables is standard when using UCT. Figure 3 below

Pure MC	INVADER	15%
MC with eval. func. based pruning	INVADER	40%
Basic hybrid UCT	INVADER	60%
Basic hybrid with improved evaluation	INVADER	67%
Hybrid UCT with all improvements	INVADER	80%
Hybrid UCT w/o parity adjustment	INVADER	73%
Hybrid UCT w/o progressive widening	INVADER	70%
INVADER with new evaluation	INVADER	38%
Hybrid UCT with all improvements	INVADER with new evaluation	89%

Fig. 3. Summary of test results

summarizes the test results where all tests were done with at least 100 games and the total time per game per player was set to 30 minutes. The first program's winning percentage over the second is shown in the third column. The first five rows show how INVADERMC performs against minimax INVADER as algorithmic features are added. The next two rows show the effect of removing one of the later enhancements. The last two rows demonstrate that the new evaluation function actually hurts the minimax INVADER, though it was a significant help for INVADERMC.

We tried a number of other techniques that seemed promising on paper but when implemented did not deliver. After UCT appeared on the Go scene and proved useful, the next big step was the use of "heavy playouts", that is, guiding the random simulations so that more natural looking random games are played [7]. All efforts to incorporate heavy playouts in INVADERMC have so far been disappointing, resulting in weaker play. Since this has had such a significant positive impact on Go-playing programs we feel that additional work in this area is critical.

INVADER has a strong and presumably reasonably accurate evaluation function. So, it seemed natural to try to eliminate completely the random simulations and simply use the evaluation function to judge the probability of winning from a given position. We were unsuccessful with this approach, suggesting that the sign of the evaluation does a good job of suggesting who is winning but converting the value of the evaluation into a winning probability is quite difficult.

It has been suggested that having the UCT tree maximize the scores rather than the winning percentages might be advantageous [10]. In Go programming, it has been shown that this is not the case [9]. Likewise, our experience with INVADERMC indicates it is best to maximize the winning percentages.

We still find it mysterious that random simulations of length 6 work so well. Adding to the mystery is the fact that a change of 2 in either direction immediately hurts the winning percentage by at least 5%. We need to understand better the mechanisms involved. We also still hope to find a way to convert or modify the evaluation function so that it can better express the actual probability of a win. An effective way of doing this will surely produce a much stronger program.

One idea to tweak the current program that might produce an improvement has to do with the UCT constant k that dictates the depth of the growing tree. It might prove beneficial to modify k as the game proceeds. For example, since later in the game there are usually fewer potential good moves we might want to change the constant to allow deeper trees to grow at the expense of possibly missing an unlikely looking move that ultimately turns out to be strong.

It is well known that self-testing is not the ideal way to test a game-playing program. However, the dearth of Amazons-playing programs, combined with the near nonexistence of publicly available programs, and the fact that the one accessible strong program known to us (AMAZONG) cannot be easily configured to play long test sets with INVADERMC make this approach necessary for now. As a kind of sanity check we registered on a game-playing Web site and entered an Amazons ladder. All moves made were those suggested by INVADERMC after approximately 10 minutes of thinking. Under this configuration INVADERMC went undefeated with a score of 9 and 0, and eventually landed at the top of the ladder. This result at least suggests that we are not going down the completely wrong path. Time constraints have prevented us from playing many games against AMAZONG (fast games are too much of a disadvantage for MC/UCT based programs), yet the few games we have played have produced a winning percentage for INVADERMC.

Acknowledgments. We want to thank the anonymous referees for their careful, thoughtful, and thorough comments and for pointing out our too frequent lapses. This paper is greatly improved as a result. We would also especially like to thank HLL for all the guidance, support, and inspiration she provided over the years.

References

1. Avetisyan, H., Lorentz, R.: Selective Search in an Amazons Program. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 123–141. Springer, Heidelberg (2003)
2. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)
3. Cazenave, T.: Iterative widening. In: In 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), pp. 523–528 (2001)
4. 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)
5. Coulom, R.: Efficient selectivity and back-up 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. Springer, Heidelberg (2007)
6. Coulom, R.: Computing "Elo Ratings" of Move Patterns in the Game of Go. ICGA Journal 30(4), 198–208 (2007)
7. Drake, P., Uurtamo, S.: Move Ordering vs Heavy Playouts: Where Should Heuristics Be Applied in Monte Carlo Go? In: Proceedings of the 3rd North American Game-On Conference (2007)
8. Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. In: Twentieth Annual Conference on Neural Information Processing Systems (2006)
9. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT for Monte-Carlo Go. Technical Report 6062, INRIA (2006)
10. Kloetzer, J., Iida, H., Bouzy, B.: The Monte-Carlo approach in Amazons. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Computer Games Workshop 2007, Amsterdam, The Netherlands, pp. 185–192 (2007)

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. Lieberum, J.: An Evaluation Function for the Game of Amazons. Theoretical Computer Science 349(22), 230–244 (2005)
13. Lorentz, R.: First-time Entry AMAZONG wins Amazons Tournament. ICGA Journal 25(3), 182–184 (2002)

Monte-Carlo Tree Search Solver

Mark H.M. Winands[1], Yngvi Björnsson[2], and Jahn-Takeshi Saito[1]

[1] Games and AI Group, MICC, Faculty of Humanities and Sciences, Universiteit Maastricht, Maastricht, The Netherlands
{m.winands,j.saito}@micc.unimaas.nl
[2] School of Computer Science, Reykjavík University, Reykjavík, Iceland
yngvi@ru.is

Abstract. Recently, Monte-Carlo Tree Search (MCTS) has advanced the field of computer Go substantially. In this article we investigate the application of MCTS for the game Lines of Action (LOA). A new MCTS variant, called MCTS-Solver, has been designed to play narrow tactical lines better in sudden-death games such as LOA. The variant differs from the traditional MCTS in respect to backpropagation and selection strategy. It is able to prove the game-theoretical value of a position given sufficient time. Experiments show that a Monte-Carlo LOA program using MCTS-Solver defeats a program using MCTS by a winning score of 65%. Moreover, MCTS-Solver performs much better than a program using MCTS against several different versions of the world-class $\alpha\beta$ program MIA. Thus, MCTS-Solver constitutes genuine progress in using simulation-based search approaches in sudden-death games, significantly improving upon MCTS-based programs.

1 Introduction

For decades $\alpha\beta$ search has been the standard approach used by programs for playing two-person zero-sum games such as chess and checkers (and many others). Over the years many search enhancements have been proposed for this framework. However, in some games where it is difficult to construct an accurate positional evaluation function (e.g., Go) the $\alpha\beta$ approach was hardly successful. In the past, Monte-Carlo (MC) methods have been used as an evaluation function in a search-tree context [1,6,7]. A direct descendent of that approach is a new general search method, called Monte-Carlo Tree Search (MCTS) [10,14]. MCTS is not a classical tree search followed by a MC evaluation, but rather a best-first search guided by the results of Monte-Carlo simulations. In the last two years MCTS has advanced the field of computer Go substantially. Moreover, it is used in other games as well (Phantom Go [8], Clobber [15]), even for games where there exists already a reasonable evaluation function (e.g., Amazons [13]). Although MCTS is able to find the best move, it is not able to prove the game-theoretic value of (even parts of) the search tree. A search method that is not able to prove or estimate (quickly) the game-theoretic value of a node may run into problems. This is especially true for sudden-death games, such as chess, that may abruptly end by the creation of one of a prespecified set of patterns

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 25–36, 2008.

[2] (e.g., checkmate in chess). In this case $\alpha\beta$ search or a special endgame solver (i.e., Proof-Number Search [3]) is traditionally preferred above MCTS.

In this article we introduce a new MCTS variant, called MCTS-Solver, which has been designed to prove the game-theoretical value of a node in a search tree. This is an important step towards being able to use MCTS-based approaches effectively in sudden-death like games (including chess). We use the game Lines of Action (LOA) as a testbed. It is an ideal candidate because its intricacies are less complicated than those of chess. So, we can focus on the sudden-death property. Furthermore, because LOA was used as a domain for various other AI techniques [5,12,20], the level of the state-of-the-art LOA programs is high, allowing us to look at how MCTS approaches perform against increasingly stronger evaluation functions. Moreover, the search engine of a LOA program is quite similar to the one of a chess program.

The article is organized as follows. Section 2 explains briefly the rules of LOA. In Sect. 3 we discuss MCTS and its application to Monte-Carlo LOA. In Sect. 4 we introduce MCTS-Solver. We empirically evaluate the method in Sect. 5. Finally, Sect. 6 gives conclusions and an outlook on future research.

2 Lines of Action

Lines of Action (LOA) [16] is a two-person zero-sum connection game with perfect information. It is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The black pieces are placed in two rows along the top and bottom of the board, while the white pieces are placed in two files at the left and right edge of the board. The players alternately move a piece, starting with Black. A move takes place in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement. A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit (i.e., the sudden-death property). In the case of simultaneous connection, the game is drawn. The connections within the unit may be either orthogonal or diagonal. If a player cannot move, this player has to pass. If a position with the same player to move occurs for the third time, the game is drawn.

3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [10,14] is a best-first search method that does not require a positional evaluation function. It is based on a randomized exploration of the search space. Using the results of previous explorations, the algorithm gradually builds up a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic steps, repeated as long as there is time left. The steps are as follows. (1) In the *selection step* the tree is traversed from the

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

3.1 The Four Strategic Steps

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

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

We use the UCT (**U**pper **C**onfidence Bounds applied to **T**rees) strategy [14], enhanced with Progressive Bias (PB [9]). UCT is easy to implement and used in many Monte-Carlo Go programs. PB is a technique to embed domain-knowledge bias into the UCT formula. It is successfully applied in the Go program MANGO. UCT with PB works as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the child k of the node p that satisfies Formula 1:

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

where v_i is the value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a coefficient, which has to be tuned experimentally. $\frac{W \times P_c}{n_i + 1}$ is the PB part of the formula. W is a constant, which has to be set manually (here $W = 100$). P_c is the *transition probability* of a move category c [17].

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

$$P_c = \frac{n_{played(c)}}{n_{available(c)}} \quad , \tag{2}$$

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

The move categories of our Monte-Carlo LOA program are similar to the ones used in the Realization-Probability Search of the program MIA [21]. They

are used in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim, and the central 2×2 board. Finally, moves are further classified based on the number of squares traveled away from or towards the center-of-mass. In total 277 move categories can occur according to this classification.

This selection strategy is only applied in nodes with visit count higher than a certain threshold T (here 50) [10]. If the node has been visited fewer times than this threshold, the next move is selected according to the *simulation strategy* discussed in the next strategic step.

Play-out. The play-out step begins when we enter a position that is not a part of the tree yet. Moves are selected in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves chosen according to a *simulation strategy*. It is well-known that the use of an adequate simulation strategy improves the level of play significantly [11]. The main idea is to play interesting moves according to heuristic knowledge. In our Monte-Carlo LOA program, the move categories together with their transition probabilities, as discussed in the selection step, are used to select the moves pseudo-randomly during the play-out.

A simulation requires that the number of moves per game is limited. When considering the game of LOA, the simulated game is stopped after 200 moves and scored as a draw. The game is also stopped when heuristic knowledge indicates that the game is probably over. The reason for doing this is that despite the use of an elaborate simulation strategy it may happen that the game-theoretical value and the average result of the Monte-Carlo simulations differ substantially from each other in some positions. In our Monte-Carlo LOA program this so-called noise is reduced by using the MIA 4.5 evaluation function [23]. When the evaluation function gives a value that exceeds a certain threshold (i.e., 1,000 points), the game is scored as a win. If the evaluation function gives a value that is below a certain threshold (i.e., -1,000 points), the game is scored as a loss. For speed reasons the evaluation function is called only every 3 plies, determined by trial and error.

Expansion. Expansion is the strategic task that decides whether nodes will be added to the tree. Here, we apply a simple rule: one node is added per simulated game [10]. The added leaf node L corresponds to the first position encountered during the traversal that was not already stored.

Backpropagation. Backpropagation is the procedure that propagates the *result* of a simulated game k back from the leaf node L, through the previously traversed node, all the way up to the root. The result is scored positively ($R_k = +1$) if the game is won, and negatively ($R_k = -1$) if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to the *value* v_L of a node. Here, it is computed by taking the average of the results of all simulated games made through this node [10], i.e., $v_L = (\sum_k R_k)/n_L$.

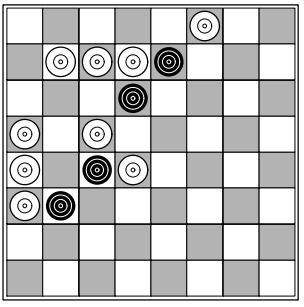


Fig. 1. White to move

4 Monte-Carlo Tree Search Solver

Although MCTS is unable to *prove* the game-theoretic value, in the long run MCTS equipped with the UCT formula is able to *converge* to the game-theoretic value. For a fixed termination game like Go, MCTS is able to find the optimal move relatively fast [25]. But in a sudden-death game like LOA, where the main line towards the winning position is narrow, MCTS may often lead to an erroneous outcome because the nodes' values in the tree do not converge fast enough to their game-theoretical value. For example, if we let MCTS analyze the position in Fig. 1 for 5 seconds, it selects **c7xc4** as the best move, winning 67.2% of the simulations. However, this move is a forced 8-ply loss, while **f8-f7** (scoring 48.2%) is a 7-ply win. Only when we let MCTS search for 60 seconds, it selects the optimal move. For a reference, we remark that it takes $\alpha\beta$ in this position less than a second to select the best move and prove the win.

We designed a new variant called MCTS-Solver, which is able to prove the game-theoretical value of a position. The backpropagation and selection mechanisms have been modified for this variant. The changes are discussed in Subsections 4.1 and 4.2, respectively. Moreover, we discuss the consequences for final move selection in Subsection 4.3. The pseudo-code of MCTS-Solver is given in Subsection 4.4.

4.1 Backpropagation

In addition to backpropagating the values {1,0,−1}, the search also propagates the game-theoretical values ∞ or $-\infty$.[1] The search assigns ∞ or $-\infty$ to a won or lost terminal position for the player to move in the tree, respectively. Propagating the values back in the tree is performed similar to negamax in the context of minimax searching in such a way that we do not need to distinguish between MIN and MAX nodes. If the selected move (child) of a node returns ∞, the node

[1] Draws are in general more problematic to prove than wins and losses. Because draws only happen in exceptional cases in LOA, we took the decision not to handle proven draws for efficiency reasons.

is a win. To prove that a node is a win, it suffices to prove that one child of that node is a win. Because of negamax, the value of the node will be set to $-\infty$. In the minimax framework it would be set to ∞. In the case that the selected child of a node returns $-\infty$, all its siblings have to be checked. If their values are also $-\infty$, the node is a loss. To prove that a node is a loss, we must prove that all its children lead to a loss. Because of negamax, the node's value will be set to ∞. In the minimax framework it would have been set to $-\infty$. In the case that one or more siblings of the node have a different value, we cannot prove the loss. Therefore, we will propagate -1, the result for a lost game, instead of $-\infty$, the game-theoretical value of a position. The value of the node will be updated according to the backpropagation strategy as described in Subsection 3.1.

4.2 Selection

As seen in the previous subsection, a node can have the game-theoretical value ∞ or $-\infty$. The question arises how these game-theoretical values affect the selection strategy. Of course, when a child is a proven win, the node itself is a proven win, and no selection has to take place. But when one or more children are proven to be a loss, it is tempting to discard them in the selection phase. However, this can lead to overestimating the value of a node, especially when moves are pseudo-randomly selected by the simulation strategy. For example, in Fig. 2 we have three one-ply subtrees. Leaf nodes B and C are proven to be a loss, indicated by $-\infty$; the numbers below the other leaves are the *expected* pay-off values. Assume that we select the moves with the same likelihood (as could happen when a simulation strategy is applied). If we would prune the loss nodes, we would prefer node A above E. The average of A would be 0.4 and 0.37 for E. It is easy to see that A is overestimated because E has more good moves.

If we do not prune proven loss nodes, we run the risk of underestimation. Especially, when we have a strong preference for certain moves (because of a bias) or we would like to explore our options (because of the UCT formula), we could underestimate positions. Assume that we have a strong preference for the first move in the subtrees of Fig. 2. We would prefer node I above A. It is easy to see that A is underestimated because I has no good moves at all.

Based on preliminary experiments, selection is here performed in the following way. In case Formula (1) is applied, child nodes with the value $-\infty$ will never be selected. For nodes of which the visit count is below the threshold, moves are selected according to the simulation strategy instead of using Formula (1). In that case, children with the value $-\infty$ *can* be selected. However, when a child with a value $-\infty$ is selected, the search is not continued at that point. The results are propagated backwards according to the strategy described in the previous subsection.

For all the children of a leaf node (i.e., the visit count equals one) we check whether they lead to a direct win for the player to move. If there is such a move, we stop searching at this node and set the node's value (negamax: $-\infty$; minimax: ∞). This check at the leaf node must be performed because otherwise it could

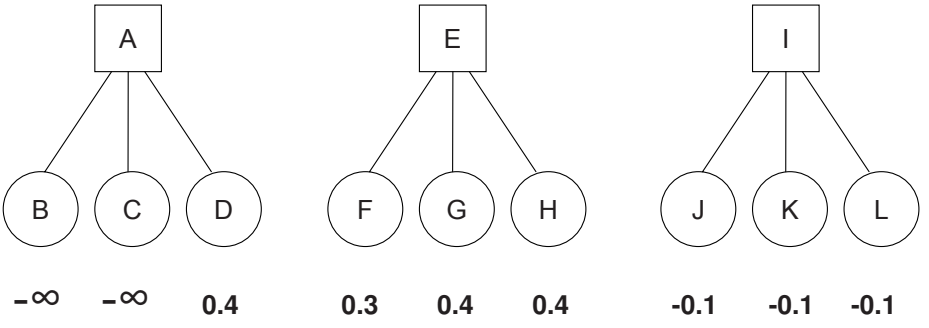


Fig. 2. Monte-Carlo Subtrees

take many simulations before the child leading to a mate-in-one is selected and the node is proven.

4.3 Final Move Selection

For standard MCTS several ways exist to select the move finally played by the program in the actual game. Often, it is the child with the highest visit count, or with the highest value, or a combination of the two. In practice, there is no significant difference when a sufficient amount of simulations for each root move has been played. However, for MCTS-Solver it *does* matter. Because of the backpropagation of game-theoretical values, the score of a move can suddenly drop or rise. Therefore, we have chosen a method called *Secure child* [9]. It is the child that maximizes the quantity $v + \frac{A}{\sqrt{n}}$, where A is a parameter (here, set to 1), v is the node's value, and n is the node's visit count.

Finally, when a win can be proven for the root node, the search is stopped and the winning move is played. For the position in Fig. 1, MCTS-Solver is able to select the best move and prove the win for the position depicted in less than a second.

4.4 Pseudo Code for MCTS-Solver

A C-like pseudo code of MCTS-Solver is provided in Fig. 3. The algorithm is constructed similar to negamax in the context of minimax search. `select(Node N)` is the selection function as discussed in Subsection 4.2, which returns the best child of the node N. The procedure `addToTree(Node node)` adds one more node to the tree; `playOut(Node N)` is the function which plays a simulated game from the node N, and returns the result $R \in \{1, 0, -1\}$ of this game; `computeAverage(Integer R)` is the procedure that updates the value of the node depending on the result R of the last simulated game; `getChildren(Node N)` generates the children of node N.

```
Integer MCTSSolver(Node N){

 if(playerToMoveWins(N))
    return INFINITY
 else (playerToMoveLoses(N))
    return -INFINITY

 bestChild = select(N)
 N.visitCount++

 if(bestChild.value != -INFINITY AND bestChild.value != INFINITY)
    if(bestChild.visitCount == 0){
        R = -playOut(bestChild)
        addToTree(bestChild)
        goto DONE
    }
    else
     R = -MCTSSolver(bestChild)
 else
  R = bestChild.value

 if(R == INFINITY){
        N.value = -INFINITY
        return R
    }
 else
    if(R == -INFINITY){

        foreach(child in getChildren(N))
          if(child.value != R){
             R = -1
             goto DONE
          }

          N.value = INFINITY
          return R
     }

  DONE:
  N.computeAverage(R)
  return R
 }
```

Fig. 3. Pseudo code for MCTS-Solver

5 Experiments

In this section we evaluate the performance of MCTS-Solver. First, we matched MCTS-Solver against MCTS, and provide results in Subsection 5.1. Next, we evaluated the playing-strength of MCTS and MCTS-Solver against different versions of the tournament LOA program MIA, as shown in Subsection 5.2. All experiments were performed on a Pentium IV 3.2 GHz computer.

5.1 MCTS vs. MCTS-Solver

In the first series of experiments MCTS and MCTS-Solver played 1,000 games against each other, playing both colors equally. They always started from the same standardized set of 100 three-ply positions [5]. The thinking time was limited to 5 seconds per move.

Table 1. 1,000-game match results

	Score	Win %	Winning ratio
MCTS-Solver vs. MCTS	646.5 - 353.5	65%	1.83

The results are given in Table 1. MCTS-Solver outplayed MCTS with a winning score of 65% of the available points. The winning ratio is 1.83, meaning that it scored 83% more points than the opponent. This result shows that the MCTS-Solver mechanism improves the playing strength of the Monte-Carlo LOA program.

5.2 Monte-Carlo LOA vs. MIA

In the previous subsection, we saw that MCTS-Solver outperformed MCTS. In the next series of experiments, we further examine whether MCTS-Solver is superior to MCTS by comparing the playing strength of both algorithms against a non-MC program. We used three different versions of MIA, considered being the best LOA playing entity in the world.[2] The three different versions were all equipped with the same latest search engine but used three different evaluation functions (called MIA 2000 [19], MIA 2002 [22], and MIA 2006 [23]). The search engine is an $\alpha\beta$ depth-first iterative-deepening search in the Enhanced Realization-Probability Search (ERPS) framework [21] using several forward pruning mechanisms [24]. To prevent the programs from repeating games, a small random factor was included in the evaluation functions. All programs played under the same tournament conditions as used in Subsection 5.1. The results are given in Table 2. Each match consisted of 1,000 games.

In Table 2 we notice that MCTS and MCTS-Solver score more than 50% against MIA 2000. When competing with MIA 2002, only MCTS-Solver is able

[2] The program won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad.

Table 2. 1,000-game match results

Evaluator	MIA 2000	MIA 2002	MIA 2006
MCTS	585.5	394.0	69.5
MCTS-Solver	692.0	543.5	115.5

to outperform the $\alpha\beta$ program. Both MC programs are beaten by MIA 2006, although MCTS-Solver scores a more respectable number of points. Table 2 indicates that MCTS-Solver when playing against each MIA version significantly performs better than MCTS does. These results show that MCTS-Solver is a genuine improvement, significantly enhancing MCTS. The performance of the Monte-Carlo LOA programs in general against MIA — a well-established $\alpha\beta$ program — is quite impressive. One must keep in mind the many man-months of work that are behind the increasingly sophisticated evaluation functions of MIA. The improvement introduced here already makes a big leap in the playing strength of the simulation-based approach, resulting in MCTS-Solver even winning the already quite advanced MIA 2002 version. Admittedly, there is still a considerable gap to be closed for MTCS-Solver before it will be a match for the MIA 2006 version. Nonetheless, with continuing improvements it is not unlikely that in the near future simulation-based approaches may become an interesting alternative in games that the classic $\alpha\beta$ approach has dominated. This work is one step towards that goal being realized.

6 Conclusion and Future Research

In this article we introduced a new MCTS variant, called MCTS-Solver. This variant differs from the traditional MC approaches in that it can prove game-theoretical outcomes, and thus converges much faster to the best move in narrow tactical lines. This is especially important in tactical sudden-death-like games such as LOA. Our experiments show that a MC-LOA program using MCTS-Solver defeats the original MCTS program by an impressive winning score of 65%. Moreover, when playing against a state-of-the-art $\alpha\beta$-based program, MCTS-Solver performs much better than a regular MCTS program. Thus, we may conclude that MCTS-Solver is a genuine improvement, significantly enhancing MCTS. Although MCTS-Solver is still lacking behind the best $\alpha\beta$-based program, we view this work as one step towards that goal of making simulation-based approaches work in a wider variety of games. For these methods, to be able to handle proven outcomes is one essential step to make. With continuing improvements it is not unlikely that in the not so distant future enhanced simulation-based approaches may become a competitive alternative to $\alpha\beta$ search in games dominated by the latter so far.

As future research, experiments are envisaged in other games to test the performance of MCTS-Solver. One possible next step would be to test the method in Go, a domain in which MCTS is already widely used. What makes this a somewhat more difficult task is that additional work is required in enabling perfect

endgame knowledge - such as Benson's Algorithm [4,18] - in MCTS. We have seen that the performance of the Monte-Carlo LOA programs against MIA in general indicates that they could even be an interesting alternative to the classic $\alpha\beta$ approach. Parallelization of the program using an endgame specific evaluation function instead of a general one such as MIA 4.5 could give a performance boost.

Acknowledgments. The authors thank Guillaume Chaslot for giving valuable advice on MCTS. Part of this work is done in the framework of the NWO Go for Go project, grant number 612.066.409.

References

1. Abramson, B.: Expected-outcome: A general model of static evaluation. IEEE Transactions on Pattern Analysis and Machine Intelligence 12(2), 182–193 (1990)
2. Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence. PhD thesis, Rijksuniversiteit Limburg, Maastricht (1994)
3. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-number search. Artificial Intelligence 66(1), 91–123 (1994)
4. Benson, D.B.: Life in the Game of Go. In: Levy, D.N.L. (ed.) Computer Games, vol. 2, pp. 203–213. Springer, New York (1988)
5. Billings, D., Björnsson, Y.: Search and knowledge in Lines of Action. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 231–248. Kluwer Academic Publishers, Boston (2003)
6. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10: Many Games, Many Challenges, pp. 159–174. Kluwer Academic Publishers, Boston (2003)
7. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)
8. Cazenave, T., Borsboom, J.: Golois Wins Phantom Go Tournament. ICGA Journal 30(3), 165–166 (2007)
9. 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)
10. 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)
11. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference on Machine Learning (ICML). ACM International Conference Proceeding Series, vol. 227, pp. 273–280. ACM, New York (2007)
12. Helmstetter, B., Cazenave, T.: Architecture d'un programme de Lines of Action. In: Cazenave, T. (ed.) Intelligence artificielle et jeux, pp. 117–126. Hermes Science (2006) (in French)
13. Kloetzer, J., Iida, H., Bouzy, B.: The Monte-Carlo Approach in Amazons. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computer Games Workshop 2007 (CGW 2007), pp. 185–192. Universiteit Maastricht, Maastricht (2007)

14. 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)
15. Kocsis, L., Szepesvári, C., Willemson, J.: Improved Monte-Carlo Search (2006), http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf
16. Sackson, S.: A Gamut of Games. Random House, New York (1969)
17. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 132–144 (2002)
18. van der Werf, E.C.D., van den Herik, H.J., Uiterwijk, J.W.H.M.: Solving Go on small boards. ICGA Journal 26(2), 92–107 (2003)
19. Winands, M.H.M.: Analysis and implementation of Lines of Action. Master's thesis, Universiteit Maastricht, Maastricht (2000)
20. Winands, M.H.M.: Informed Search in Complex Games. PhD thesis, Universiteit Maastricht, Maastricht (2004)
21. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)
22. Winands, M.H.M., Kocsis, L., Uiterwijk, J.W.H.M., van den Herik, H.J.: Temporal difference learning and the Neural MoveMap heuristic in the game of Lines of Action. In: Mehdi, Q., Gough, N., Cavazza, M. (eds.) GAME-ON 2002, Ghent, Belgium, pp. 99–103. SCS Europe Bvba (2002)
23. Winands, M.H.M., van den Herik, H.J.: MIA: a world champion LOA program. In: The 11th Game Programming Workshop in Japan (GPW 2006), pp. 84–91 (2006)
24. Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., van der Werf, E.C.D.: Enhanced forward pruning. Information Sciences 175(4), 315–329 (2005)
25. Zhang, P., Chen, K.: Monte-Carlo Go tactic search. In: Wang, P., et al. (eds.) Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007), pp. 662–670. World Scientific Publishing Co. Pte. Ltd, Singapore (2007)

An Analysis of UCT in Multi-player Games

Nathan R. Sturtevant

Department of Computing Science, University of Alberta,
Edmonton, AB, Canada, T6G 2E8
nathanst@cs.ualberta.ca

Abstract. The UCT algorithm has been exceedingly popular for Go, a two-player game, significantly increasing the playing strength of Go programs in a very short time. This paper provides an analysis of the UCT algorithm in multi-player games, showing that UCT, when run in a multi-player game, is computing a mixed-strategy equilibrium, as opposed to max^n, which computes a pure-strategy equilibrium. We analyze the performance of UCT in several known domains and show that it performs as well or better than existing algorithms.

1 Introduction

Monte-Carlo methods have become popular in the game of Go over the last few years, and even more so with the introduction of the UCT algorithm [3]. Go is probably the best-known two-player game in which computer players are still significantly weaker than humans. UCT works particularly well in Go for several reasons. First, in Go it is difficult to evaluate states in the middle of a game, but UCT only evaluates endgames states, which is relatively easy. Second, the game of Go converges for random play, meaning that it is not very difficult to get to an end-game state.

Multi-player games are also difficult for computers to play well. First, it is more difficult to prune in multi-player games, meaning that normal search algorithms are less effective at obtaining deep lookahead. While alpha-beta pruning reduces the size of a game tree from $O(b^d)$ to $O(b^{d/2})$, the best techniques in multi-player games only reduce the size of the game tree to $O(b^{\frac{n-1}{n}d})$, where n is the number of players in the game [6]. A second reason why multi-player games are difficult is because of opponent modeling. In two-player zero-sum games opponent modeling has never been shown to be necessary for high-quality play, while in multi-player games, opponent modeling is a necessity for robust play versus unknown opponents in some domains [9].

As a result, it is worth investigating UCT to see how it performs in multi-player games. We first present a theoretical analysis, where we show that UCT computes a mixed-strategy equilibrium in multi-player games and discuss the implications of this. Then, we analyze UCT's performance in a variety of domains, showing that it performs as well or better as the best previous approaches.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 37–49, 2008.

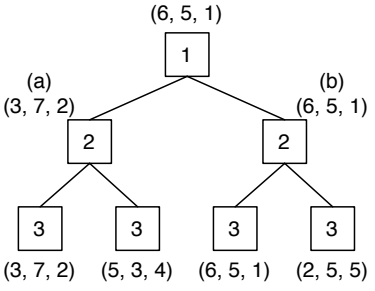


Fig. 1. A sample max^n tree

2 Background

The max^n algorithm [4] was developed to play multi-player games. Max^n searches a game tree and finds a strategy which is in equilibrium. That is, if all players were to use this strategy, no player could unilaterally gain by changing their strategy. In every perfect information extensive form game (e.g., tree search) there is guaranteed to be at least one pure-strategy equilibrium, that is, one that does not require the use of mixed or randomized strategies.

We demonstrate the max^n algorithm in Fig. 1, a portion of a 3-player max^n tree. Nodes in the tree are marked with the player to move at that node. At the leaves of the tree each player's payouts are in a n-tuple, where the ith value is the payoff for the ith player. At internal nodes in a max^n tree the player to play selects the move that leads to the maximum payoff. So, at the node marked (a), Player 2 chooses (3, 7, 2) to get a payoff of 7 instead of (5, 3, 4) to get a payoff of 3. At the node marked (b) Player 2 can choose either move, because they both lead to the same payoff. In this case Player 1 chooses the leftmost value and returns (6, 5, 1). At the root of the tree Player 1 chooses the move which leads to the maximum payoff, (6, 5, 1).

While the max^n algorithm is simple, there are several complications. In real games players rarely communicate explicitly before the beginning of a game. This means that they are not guaranteed to be playing the same equilibrium strategy, and, unlike in two-player games, max^n does not provide a lower bound on the final payoff in the game when this occurs [8]. In practice it is not always clear which payoffs should be used at leaf nodes. The values at the leaves of a tree may be scores, but can also be the utility of each payoff, where the opponents' utility function is not known *a priori*. For instance, one player might play a riskier strategy to increase the chances of winning the game, while a different player may be content to take second place instead of risking losing for the chance of a win. While the first approach may be better from a tournament perspective [1], you cannot guarantee that your opponents will play the best strategies possible. This might mean, for instance, that in Fig. 1 Player 2 has a different preference at node (b). If Player 2 selects (2, 5, 5) at node (b), Player 1 should choose to move to the left from the root to get (3, 7, 2).

Two algorithms have been introduced that attempt to deal with imperfect opponent models. The soft-max^n algorithm [9] uses a partial ordering over game outcomes to analyze games. It returns sets of max^n values at each node, with each max^n value in the set corresponding to a strategy that the opponents might play in a subtree. The prob-max^n algorithm [5] uses sets of opponent models and with probabilistic weights which are used for back-up at each node according to current opponent models. Both algorithms have learning mechanisms for updating opponent models during play. In the game of Spades, these approaches were able to mitigate the problems associated with an unknown opponent. We will discuss these results more in our experimental section.

2.1 UCT

UCT [3] is one of several recent Monte-Carlo-like algorithms proposed for game-playing. The algorithm plays games in two stages. In the first stage a tree is built and explored. The second stage begins when the end of the UCT tree is reached. At this point a leaf node is expanded and then the rest of the game is played out according to a random policy. The UCT tree is built and played out according to a greedy policy. At each node in the UCT tree, UCT selects and follows the move i for which

$$\bar{X}_i + C\sqrt{\frac{\ln T}{T_i}}$$

is maximal, where $\bar{X}_i$ is the average payoff of move i, T is the number of times the parent of i has been visited, and T_i is the number of times i has been sampled. C is a tuning constant used to trade off exploration and exploitation. Larger values of C result in more exploration. In two-player games the move/value returned by UCT converges on the same result as minimax.

3 Multi-player UCT

The first question we address is what computation is performed by UCT on a multi-player game tree. For this analysis we assume that we are searching on a finite tree and that we can perform an unlimited number of UCT samples. In this limit the UCT tree will grow to be the size of the full game tree and all leaf values will be exact.

Returning to Fig. 1 we can look to see what UCT would compute on this tree. At node (a) Player 2 will always get a better payoff by taking the left branch to get (3, 7, 2). In the limit, for any value of C, the value at node (a) will converge to (3, 7, 2). At branch (b), however, both moves lead to the same payoff. Initially, they will both be explored once and have the same payoff. On each subsequent visit to branch (b), the move which has been explored least will be explored next. As a result, at the root of the sub-tree rooted at (b) will return (6, 5, 1) half of the time and (2, 5, 5) the other half. The average payoff of the move towards node (b) will then be (4, 5, 3). The resulting strategy for the entire tree is for

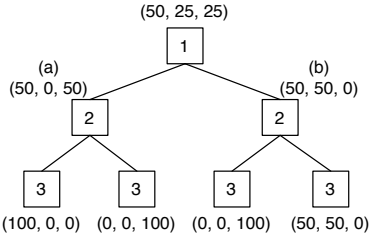


Fig. 2. Mixed equilibrium in a multi-player game tree

the player at the root to move to the right towards node (b) to get an expected payoff of 4.

The final strategy is mixed, in that Player 2 is expected to randomize at node (b). Playing a mixed strategy makes the most sense in a repeated game, where over time the payoffs will converge due to the mixing of strategies. But, in a one-shot game this makes less sense, since a player cannot actually receive the equilibrium value. Many games are repeated, however, although random elements in the game result in different game trees in each repetition (e.g., from the dealing of cards in a card game). In this context randomized strategies still make sense, as the payoffs will still average out over all the hands. Randomized strategies can also serve to make a player more unpredictable, which will make the player harder to model.

Theorem 1. UCT computes an equilibrium strategy, which may be mixed, in a multi-player game tree.

Proof. We have demonstrated above that in some trees UCT will produce a mixed strategy. In the limit of infinite samples UCT returns a strategy that is in equilibrium in a finite tree because it selects the maximum possible value at each node in the UCT tree. This means that there is no other move that the player could choose at this node to unilaterally improve their payout. □

UCT will always compute an evenly mixed strategy at the leaves of the tree. But, there is no guarantee that in practice it will compute the expected mixed strategy at internal nodes in the tree. If ties are always broken from left to right, in Fig. 2 the value of node (a) for Player 1 will converge on 50 from above, while the value of node (b) for Player 1 will converge on 50 from below. As a result, depending on the number of samples and the value of C, UCT may not end up evenly mixing strategies higher up in the game tree. The max^n algorithm will never mix strategies like UCT, although it is not hard to modify it to do so. But, the biggest strength of UCT is not its ability to imitate max^n, but its ability to infer an accurate evaluation of the current state-based samples of possible endgame states.

4 Experiments

Given the theoretical results presented in the last section, we run a variety of experiments to compare the performance of UCT to previous state-of-the-art programs in three domains. These experiments cover a variety of game types and give insight into the practical performance of UCT in multi-player games.

Most experiments reported here are run between two different player types (UCT and an existing player) in either a 3-player or 4-player game. In order to reduce the variance of the experiments, they are repeated multiple times for each possible arrangement of player types in a game. For instance, in a 3-player game there are eight ways to arrange two players. Two of these arrangements contain all of a single type of player, so we remove these arrangements, resulting in the configurations shown in Table 1. Playing duplicate matches in this way makes it easier to measure statistical significance.

4.1 Chinese Checkers

Our first experiments are in the game of Chinese Checkers. A Chinese Checkers board is shown in Fig. 3. In this game the goal is to get your pieces across the board and into a symmetric position from the start state as quickly as possible. Chinese Checkers can be played with anywhere from two to six players. We experiment with the three-player version here. Chinese Checkers can be played on different

Table 1. 6 ways to arrange two player types in a 3-player game

Player 1	Player 2	Player 3
Type A	Type A	Type B
Type A	Type B	Type A
Type A	Type B	Type B
Type B	Type A	Type A
Type B	Type A	Type B
Type B	Type B	Type A

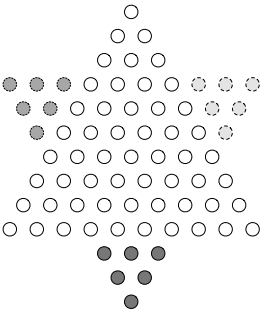


Fig. 3. A 3-player Chinese Checkers board

Table 2. Experiments in the game of Chinese Checkers

UCT (Random)	100	500	2500	10000	50000
Average Distance	9.23	4.21	2.08	1.95	2.29
Wins	1%	13%	43%	44%	34%
UCT (e-greedy)	100	500	2500	10000	50000
Average Distance	4.12	2.36	1.63‡	1.43†	0.90
Wins	14%	33%	42%	50%	73%

sized boards with different numbers of checkers. We use a slightly smaller board than is most commonly used, because this allows us to use a stronger evaluation function. This evaluation function for the opponent, a max^n player, is based on a lookup table containing the optimal distance from any state to the goal state given that no other players' pieces are on the board. A player using this evaluation function will play strong openings and perfect endgames. Because the evaluation function ignores the opponents' pieces, it is not quite as strong during the mid-game phase when the middle of the board can get congested.

We used two versions of UCT for these experiments. For both versions, we disallowed backwards moves; otherwise the game will not progress and the randomly sampling fails. The first version of UCT uses purely random playouts. The second version uses an epsilon-greedy playout policy that takes the best greedy move (the one that moves a piece the farthest distance) with 95% probability, and plays randomly 5% of the time. We compared these policies by playing 100 games in each of the configurations from Table 1. (Thus there were 600 total games played with each player type playing 900 times.)

When comparing these policies, given 500 samples each, the epsilon-greedy policy won 81% of the games. When the game ended, the epsilon-greedy player was, on average, 0.94 moves away from the goal state. In contrast, the random-playout player was 2.84 moves away from the goal state. The player that wins is averaged in as 0 moves away from the goal state. These distances were computed using the max^n player's evaluation function.

A comparison of the two UCT players against the max^n player is found in Table 2. In these experiments C was fixed at 4.0, and we varied the number of samples that UCT was allowed, recording the percentage of wins against the max^n player as well as the average distance from the goal at the end of the game. The player with random playouts never won more than 44% of the games played. The epsilon-greedy player, however, performed on a par with the max^n player when given 10,000 samples, and beat it by a large margin when given 50,000 samples. In both of these situations, the UCT player was closer to the goal at the end of the game. All results are statistically significant with 99% confidence except for those marked † which are only significant at 95%. The results marked ‡ are not statistically significant.

It is interesting to note that the epsilon-greedy player's performance monotonically increases as we increase the number of playouts, but the random-playout player does not exhibit this same tendency. In particular, the player with 50k

simulations plays markedly worse than the player with 10k simulations. It seems that this occurs because there are so many bad moves that can be made during random playout. Thus, the random playouts are not reflective of how the game will actually be played, and may lead towards positions which are likely to be won in random play, but not in actual play.

In these experiments the UCT player has a computation advantage over the max^n player, which only looks ahead 4-ply, although the max^n player has a significant amount of information available in the evaluation function. To make this more concrete, at the start of the game there are 17,340 nodes in the 4-ply tree given pruning in max^n, but there are 61 million nodes in the 7-ply tree. By the fourth move of the game there are 240 million nodes in the 7-ply tree. This is important because the only significant gains in performance in a three-player game come with an additional 3-ply of search, when the first player in the search can see an additional[1] move of his own moves ahead. On a 2.4Ghz Intel Core 2 Duo, we can expand 250k nodes per second, so a 7-ply search takes about 1000 seconds. Our UCT implementation plays about 900 games per second, so it takes about a minute to do 50k samples. Thus, while the UCT computation is more expensive than the max^n computation, the max^n player will not see an improvement in playing strength unless it is allowed significantly more time to search.

If we were looking to refine the player here, the database lookups used by the max^n player could be used as an endgame database by the UCT player and might further increase its strength or speed. Regardless, we have shown that UCT is quite strong in the game of Chinese Checkers, when given sufficient time for play.

4.2 Spades

Our next experiments are in the card game Spades. A game in Spades is divided into multiple hands which are played semi-independently. The first player to reach 300 points over all hands wins. In Spades players bid on the number of tricks which they expect to take. There is a large, immediate penalty for taking fewer tricks than bid. If, over a period of time, a player takes too many extra tricks (overtricks), there is also a large penalty. Thus, the best strategy in the game is to ensure that you make your bid, but then to avoid extra tricks after that.

Previous work in Spades [5,9] demonstrated that the selection of an opponent model is very important. A superior strategy may only be better if you have an accurate opponent model. We duplicate these experiments here to demonstrate that while UCT is strong, it does not avoid the need for opponent modeling. Then, we compare the prob-max^n algorithm to UCT.

These experiments were played open-handed so that all players could see other players' cards. In real play, we can sample many possible opponent hands and solve them individually to find the best move, as has been done in Bridge [2]. But, in our experiments, open-handed results have always been highly correlated with a sampling approach, so we only perform the open-handed experiments, which are much cheaper.

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

Table 3. UCT performance in Spades

	Algorithm 1	Algorithm 2	Avg. Score Alg. 1	Avg. Score Alg. 2	Algorithm 1 win %
A	mOT_{mOT}	mOT_{mOT}	245.91	-	-
B	MT_{MT}	MT_{MT}	202.71	-	-
C	mOT_{MT}	MT_{mOT}	231.84	171.48	67%
D	mOT_{MT}	MT_{MT}	214.33	209.30	51.5%
E	mOT_{mOT}	MT_{mOT}	203.72	188.96	55%
F	mOT_{mOT}	MT_{MT}	179.19	212.76	43%
G	mOT_{gen}	mOT_{MT}	243.14	238.65	51%
H	mOT_{gen}	mOT_{mOT}	240.06	245.70	48.5%
I	mOT_{gen}	MT_{mOT}	235.41	181.54	64%
J	mOT_{gen}	MT_{MT}	215.72	207.17	51.5%
K	mOT_{gen}	prob-max^n	214.58	198.21	52.8%
L	mOT_{gen}	prob-max^n (learn)	212.60	202.67	53.2%

Each player was given 7 cards, and the game was played with 3 players, so the total depth of the tree was 21 ply. This means that the full game tree can be analyzed by prob-max^n. Again, each game was repeated multiple times according to the configurations in Table 1. There are two player types in these experiments. The mOT player tries to make their bid and minimize overtricks (**m**inimize **O**ver**T**ricks). The MT player attempts to maximize the number of tricks taken (**M**aximize **T**ricks), irrespective of the bid. Player types are subscripted by the opponent model that is being used. So, mOT_{MT} is a player that tries to minimize their overtricks and believes that their opponents are trying to maximize the number of tricks they are taking.

Experimental results for Spades are in Table 3. All experiments are run with UCT doing 10,000 samples per move with $C = 2.0$. There are no playout rules; all playouts beyond the UCT tree are purely random. Lines A and B show the performance of each player when playing against itself. mOT players average 245.91 points per game, clearly better than the MT player which only averages 202.71 points per game.

Lines C-F demonstrate what happens when a mOT player has correct (C-D) or incorrect (E-F) opponent models. The most interesting line here is line F. When mOT has the wrong opponent model and MT does as well, mOT has a lower average score than MT and only wins 43% of the games. As stated previously, these results have been observed before with max^n. These experiments serve to confirm these results and show that, in this situation, the mixed equilibrium computed by UCT is not inherently better than the equilibrium computed by max^n. The discrepancy in performance can be partially resolved by the use of generic opponent models, which just assume that the opponent is trying to make their bid, but nothing else. Using the model, in lines G-J, the mOT_{gen} player beats every opponent except a mOT_{mOT} opponent.

Finally, in lines K-L we compare UCT using the mOT strategy and a generic opponent model to prob-max^n. In line K the prob-max^n player does not do any learning, while in line L it does. The results in line K are significant with 99%

Table 4. Detailed comparison of UCT and prob-max^n

	Algorithm	Player 1 Avg	Player 2 Avg	Player 3 Avg
K	mOT_{gen}	211.7	219.2	212.8
K	prob-max^n	227.2	220.3	147.1
L	mOT_{gen}	208.6	218.0	211.2
L	prob-max^n (learn)	227.7	228.1	152.3

confidence, but in line L they are only significant with 95% confidence. At first glance, it seems that the UCT player is better than prob-max^n, but this is not entirely true.

When breaking down these results into per-player performance, we notice an interesting trend. The player which plays last (the third player) almost always scores worse than the other players. This is the case in lines A-J for the UCT player and in K-L for the prob-max^n player. There are two reasons why this is not surprising. The player who moves last has extra constraints on their possible bids, meaning they may be forced to under- or over-bid. The player moving last also has less control over the game.

We break-down the results by player position in Table 4. Here, we can see what is happening more clearly. The prob-max^n player outplays UCT when playing as Player 1 with 99% confidence or Player 2 by a small margin, but loses badly as Player 3. As stated before, this is not a feature of prob-max^n, but of other algorithms as well. So, the novel aspect of this situation is that the UCT player manages to avoid playing poorly against prob-max^n when in the third position. We do not have a systematic explanation of this effect, but have noticed that these types of inconsistencies are common in multi-player games. We are working on further analysis.

One may be tempted to think that the comparison here has been bogged down by opponent modeling issues. In fact, the opposite is true. The experiments show that opponent modeling *is* the issue in many multi-player domains, especially one with such sharp changes in the evaluation function, as is the case in Spades.

4.3 Hearts – Shooting the Moon

Our third domain is the card game, Hearts. The goal of Hearts is to take as few points as possible. Like Spades, a game of Hearts is made up of multiple hands; a game ends when a player's score reaches or exceeds 100 points, and the player with the lowest score wins. We experiment on the 4-player version of Hearts here, which is most common. We play these games with all the cards face up, for the same reasons as in Spades.

One of the difficulties of Hearts is that there are two conflicting ways to play the game. The normal goal is to take as few points as possible. But, if a player manages to take all the points, called 'shooting the moon', this player will get 0 points instead, and the other players will get 26 each. Thus, good players are willing to take a few points to keep other players from shooting the moon.

Table 5. Shooting the moon in Hearts

Algorithm Comparisons

	UCT	Self-trained	UCT-trained	Random play	Simple max^n
total	250	312	362	411	1377
perc.	7.70%	9.62%	11.16%	12.67%	42.45%

UCT Parameter Comparisons

	0.0	0.2	0.4	0.6	0.8	1.0
total	444	310	285	292	315	332
perc.	13.69%	9.56%	8.79%	9.00%	9.71%	10.23%

	0.0 / 0.4	0.2 / 0.4	0.0 / 0.6	0.2 / 0.6	0.4 / 0.6
total	250	285	273	298	303
perc.	7.70%	8.79%	8.42%	9.19%	9.34%

To measure the skill of different players in this aspect of the game, we created a library of games in which one player could possibly shoot the moon. These games were found by playing 10,000 games with two player types. In these games the ith player would try to take all the points, and the remaining players would try to avoid taking points. Out of 40,000 possibilities (10,000 games times 4 possible hands in each game) we found 3,244 hands in which a player might be able to shoot the moon. We then ran a set of different algorithms against this shooting player in the 3,244 hands. In Table 5 we report how many times the opponent was able to shoot the moon against each search algorithm.

A simple max^n player which does not explicitly attempt to stop the opponents from shooting was only able to stop the opponent from shooting 1867 times, leaving 1377 times when the opponent shot. A random player was able to stop the opponent from shooting in all but 411 games. This player does not play hearts well, but as a result is able to disrupt the normal play of the game. Two players which learned to play hearts through self-play and play against UCT stopped the opponents from shooting in all but 312 and 362 cases, respectively. The best algorithm in this comparison was UCT, which only had the opponent shoot 250 times when using 50,000 samples.

We experimented with a wide variety of UCT parameters here, and we summarize the results in the bottom part of Table 5. First, we varied C from 0.0 to 1.0 by increments of 0.2. In these experiments, values of 0.4 and 0.6 produced the best performance, stopping all but 285 and 292 situations, respectively. We then tried a new approach, where we used one value of C for all nodes where the player at the root is to play, and used a different value at all other nodes. The intuition behind this approach is that a player should quickly find the best move for itself, but explore the opponents' responses more thoroughly. Using a value of 0.0 for the player at the root and 0.4 for other players produced the best behavior, better than the results produced when either 0.0 or 0.4 was used for all players.

This experiment is interesting because UCT has to find the specific line of play an opponent might use to shoot the moon in order to stop it. Using a lower value of C increases the depth of the UCT tree, which helps ensure that a safe

line of play is found. However, this does not necessarily guarantee that the best line of play is found. In some sense this can randomize play slightly without too large a penalty.

4.4 Hearts – Quality of Play

To test the quality of play, we played repeated games of Hearts. That is, a given configuration of players played multiple hands of Hearts until one player's score reached 100. The final score for an algorithm is the average score of all players using that algorithm at the end of the game. We also report the standard deviation of the final score, as well as the percentage of games in which a given player had the highest score at the end of the game. Experimental results are in Table 6. All results are statistically significant with 99% confidence.

The current state-of-the-art players in Hearts use a learned evaluation function [10]. We trained three players to play Hearts using methods similar to, but more efficient than those described in [10]. Whether we trained through self-play or against UCT players of varying strength, the learned players had essentially identical performance in the full game, scoring 20 points more on average than the UCT player. (Lower scores are better.) The results were stable both for the different learned players and against UCT with up to 50,000 samples per turn; we report results against UCT with 5000 samples. For comparison, an older, hand-tuned, player [7] averaged 88.31 points per game against UCT, just better than the random player, which averaged 89.23 points a game. However, UCT's average score against the random player, 16.31, is much better than against the hand-tuned player, 51.77.

In the bottom part of Table 6 we experiment with different numbers of playouts. Here we played UCT with k playouts against UCT with $2k$ playouts. The player with more playouts consistently averages 4-5 points better than the player with fewer playouts. This shows that UCT does improve its performance as the number of simulations increases.

In all these experiments, UCT is doing purely random playouts. We experimented with various playout policies, but there were no simple policies which

Table 6. Performance of UCT against various opponents

	Learned	Hand-tuned	Random
UCT 5000	46.12 (30.6)	51.77 (27.2)	16.31 (13.7)
Opponent	67.30 (43.1)	88.31 (24.5)	89.23 (24.1)
loss perc.	83.9%	88.0%	100%

	$k = 100$	250	500	1000	2000	4000	8000
UCT k	79.46	77.69	77.18	76.49	76.25	76.47	76.59
(std dev)	(26.95)	(26.78)	(27.07)	(27.38)	(26.65)	(27.06)	(26.76)
loss perc.	66.4%	59.4%	58.7	57.1%	55.5%	58.3%	56.7%
UCT $2k$	67.07	69.92	71.40	71.52	72.04	71.91	72.17
(std dev)	(27.69)	(27.86)	(27.35)	(27.63)	(27.11)	(26.84)	(26.80)

increased the strength of play. For instance, a simple policy for play is to play always the highest card that will not win the trick. As a game-playing policy, this is stronger than random play, but makes a poorer playout module than random.

In these experiments we used a value of $C = 0.4$. When compared to the alternating values of C used in the previous section $C = 0.4$ provided the best performance. Here is to remark that both players always beat the learned players. The learned players tested are not nearly as aggressive about attempting to shoot the moon as the player in the last section. This means that the 'insurance' paid to keep one's opponent from shooting the moon is less likely to payoff in these experiments.

5 A Summary of Findings and Future Work

This paper provides a foundation for future work on UCT in multi-player games. It shows theoretically that UCT computes a mixed-strategy equilibrium, unlike the traditional computation performed by max^n. UCT, given a strong playout policy and sufficient simulations, is able to beat a player that uses a very strong evaluation function in the game of Chinese Checkers. In other domains, UCT plays on a par with existing programs in the game of Spades, and slightly better than existing programs in Hearts.

These results are promising and suggest that UCT has the potential to richly benefit multi-player game-playing programs. However, UCT in itself is not a panacea. UCT does not offer any solution to the problem of opponent modeling, and will need to be combined with opponent-modeling algorithms if we want to achieve expert-level play in these programs.

Additionally, the card games we experimented with in this paper were all converted into perfect-information games for the experiments here. We are continuing to work on issues related to imperfect information. For instance, UCT can be modified to handle some situations in imperfect information games that cannot be solved by the simple approaches discussed here, but the exact application of these techniques is still a matter of future research.

References

1. Billings, D.: On the importance of embracing risk in tournaments. ICGA Journal 29(4), 199–202 (2006)
2. Ginsberg, M.: Gib: Imperfect information in a computationally challenging game. Journal of Articial Intelligence Research 14 (2001)
3. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Proceedings of the 17th European Conference on Machine Learning, pp. 282–293. Springer, Heidelberg (2006)
4. Luckhardt, C., Irani, K.: An algorithmic solution of N-person games. In: AAAI 1986, vol. 1, pp. 158–162 (1986)
5. Sturtevant, N., Zinkevich, M., Bowling, M.: Probmaxn: Opponent modeling in n-player games. In: AAAI 2006, pp. 1057–1063 (2006)

6. Sturtevant, N.R.: Last-branch and speculative pruning algorithms for max^n. In: IJCAI 2003, pp. 669–678 (2003)
7. Sturtevant, N.R.: Multi-Player Games: Algorithms and Approaches. PhD thesis, Computer Science Department, UCLA (2003)
8. Sturtevant, N.R.: Current challenges in multi-player game search. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 285–300. Springer, Heidelberg (2006)
9. Sturtevant, N.R., Bowling, M.H.: Robust game play against unknown opponents. In: AAMAS 2006 (2006)
10. Sturtevant, N.R., White, A.M.: Feature construction for reinforcement learning in hearts. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 122–134. Springer, Heidelberg (2007)

Multi-player Go

Tristan Cazenave

LIASD, Université Paris 8, 93526, Saint-Denis, France
cazenave@ai.univ-paris8.fr

Abstract. Multi-player Go is Go played with more than two colors. Monte-Carlo Tree Search is an adequate algorithm to program the game of Go with two players. We address the application of Monte-Carlo Tree Search to multi-player Go.

1 Introduction

The usual algorithm for multi-player games is $\max^n$ [11,13]. In this contribution, we propose alternative UCT (UCT stands for Upper Confidence bounds applied to Trees) based algorithms for multi-player games. We test the algorithms on multi-player Go.

We start admitting that two-player Go is already a complex game which is difficult to program [1]. However, recent progress in Monte-Carlo Tree Search makes it easier and gives better results than previous algorithms. Going from two-player Go to multi-player Go makes the game more complex. Yet, the simplicity and the strength of Monte-Carlo Tree Search algorithms may help to manage multi-player Go, and may result in effective programs.

The course of this paper is as follows. Section 2 presents recent research on Monte-Carlo Tree Search. Section 3 explains some subtleties of multi-player Go. Section 4 details the different algorithms that can be used to play multi-player Go. In Sect. 5 we give experimental results. Section 6 provides a summary of findings.

2 Monte-Carlo Tree Search

This section gives an overview of Monte-Carlo Tree Search. First we discuss search and Monte-Carlo Go (2.1) and then we explain the RAVE algorithm (2.2).

2.1 Search and Monte-Carlo Go

Monte-Carlo Go started as a simulated annealing on the list of possible moves [2]. Then a straightforward sampling method consisting of playing random playouts replaced it. Nowadays, the programs develop a tree before they start the playouts [5,8]. MOGO [6,7] and CRAZY STONE [5] are good examples of the success of these methods. The UCT algorithm [8] is currently the standard algorithm used for Monte-Carlo Go programs.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 50–59, 2008.

2.2 RAVE

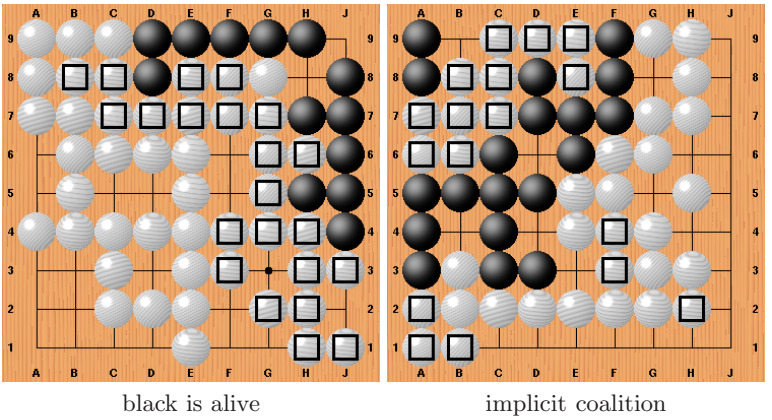
The RAVE (Rapid Action Value Estimation) algorithm [6] improves on UCT by using a rapid estimation of the value of moves when the number of playouts of a node is low. It uses a constant k and a parameter β that progressively switches from the rapid estimation heuristic to the normal UCT value. The parameter β is computed using the formula: $\beta = \sqrt{\frac{k}{3 \times games + k}}$. β is then used to bias the evaluation of a move in the tree with the formula: $val_i = \beta \times heuristic + (1.0 - \beta) \times UCT$.

The rapid estimation consists in computing statistics on all possible moves at every node of the UCT tree. After each playout, every move in the node, which has the same color in the node and in the playout is updated with the result of the playout. For example, the last move of a playout is used to update the statistics of the corresponding move in the nodes of the UCT tree that start the playout. The value of the heuristic is the mean value of the move computed using these statistics. It corresponds to the mean result of the playouts where the move has been played.

3 Multi-player Go

Multi-player Go is sometimes played for fun at Go tournaments or at Go clubs. In this section we explain some subtleties that are specific to multi-player Go. We use three colors, Black, White, and Red, who play in that order. In the figures, Red stones are the stones marked by squares.

The first board of Fig. 1 shows a particular way to live in three-player Go. The black stones are alive, provided Black does not play at H8. White cannot play at H8, and if Red plays at H8 and captures the White stone, Black immediately recaptures the Red stone and gets two eyes.



black is alive implicit coalition

Fig. 1. Some particularities of multi-player Go

The second board shows an implicit coalition. It is White to play. In normal two-player Go, the semeai between Black and Red in the upper left corner is won by Black. However, here it is profitable to White to kill Black as it will enable him to expand its territory, and it is of course also profitable to Red to kill Black as it will give him chances to live. So, in this situation, even if Black is better in its semeai against Red, it will lose it because White will play at G8, and Red will play at D6 just after and capture Black.

If we define for each player that the goal of multi-player Go is to be the winner of the game, we define a queer game [10], since it means that in some positions, no player can force a win. A main problem with this definition is that a player who is losing in the game can decide who will be the winner. For example, in three-player Go, if a player P1 has a small group surrounded by player P2, and if P2 wins if the small group is captured, but loses if the small group lives, P1 can either always pass and let P2 win, or make his group alive and let P3 win. When we implemented and tested three-player Go with the above definition, similar situations often happened and P2 was always passing because all the moves were losing moves, letting the weaker player win.

To avoid queer games, we decided to define the goal of multi-player Go as to score as many points as possible, using Chinese rules. Every stone of a player counts as one point in the end of a game. Moreover, every empty intersection completely surrounded by only one player also counts as one point. A game ends when all the players have consecutively passed.

A second possibility would have been to use Japanese rules and to remove a string solely if it is surrounded by one color only. Not removing a string surrounded by more than one color would be a very strong bias on the game since it would make weak surrounded strings alive, and the game would be much different from usual Go. Using Chinese rules is natural for Monte-Carlo based programs and Japanese rules would be even more complicated for multi-player Go.

4 Multi-player Algorithms

In this section, we describe different algorithms for multi-player games. We start with $\max^n$, the algorithm currently used in most multi-player games. Then we briefly discuss UCT, Coalitions of players, Paranoid UCT, UCT with alliance, and eventually variations on Confident UCT.

4.1 Max^n

The usual algorithm for multi-player games is $\max^n$ [13] which is the extension of Minimax to multi-player games. Some pruning techniques can be used with $\max^n$ [9,12]. We did not test this algorithm on Go for two reasons: (1) building an evaluation function for Go is difficult and (2) the UCT framework is straightforward and powerful in itself.

4.2 UCT

Multi-player UCT behaves the same as UCT except that instead of having one result for a playout, the algorithm uses an array of scores with one result for each player which is the number of points of each player at the end of the playout. When descending the UCT tree, the player uses the mean result of its scores, not taking into account the scores of the other players.

4.3 Coalitions of Players

In multi-player games, in contrast to two-player games, players can collaborate and form coalitions. In order to model a coalition between players, we have used a simple rule: it consists in not filling an empty intersection that is surrounded by players of the same coalition, provided that none of the surrounding strings is in atari. Of course, this rule only applies for players of the same coalition.

We also defined the Allies' scoring algorithm which is different from the Normal scoring algorithm. In Normal scoring, an empty intersection counts as a point for a player if it is completely surrounded by stones of the player's color. In Allies' scoring, an empty intersection is counted as territory for a player if it is surrounded by stones that can be either of the color of the player or of the color of any of its allies.

The third scoring algorithm we have used is the Joint scoring algorithm, it consists in counting allied eyes as well as allied stones as points for players in a coalition.

4.4 Paranoid UCT

The paranoid algorithm consists in considering that all the other players form a coalition against the player. The UCT tree is traversed in the usual way, but the playouts use the coalition rules for the other players.

The algorithm has links with the paranoid search of Sturtevant and Korf [14] as it considers that all the other players are opponents.

In the design of the Paranoid UCT, we choose to model paranoia modifying the playouts. A second possibility is to model it directly in the UCT tree. In this case, the other players choose the move that minimizes the player mean instead of choosing the move that maximizes their own mean when descending the UCT tree. It could also be interesting to combine the two approaches.

4.5 UCT with Alliances

The alliance algorithm models an explicit coalition of players. The playouts of the players in the alliance use the coalition rule. The playouts can be scored either using the Normal, the Allies', or the Joint scoring algorithm.

4.6 Confident UCT

Confident UCT algorithms dynamically form coalitions depending on the situation on the board. Below we discuss three different types of confident algorithms.

The first one is the Confident algorithm. At each move it develops as many UCT trees as there are players. Each UCT tree is developed assuming a coalition with a different player. This includes developing a UCT tree with a coalition with itself, which is equivalent to the normal UCT algorithm. Among all these UCT trees, it chooses the one that has the best mean result at the root, and plays the corresponding move. The idea of the confident algorithm is to assume that the player, who is the most interesting to cooperate with, will cooperate. In case of the paranoid algorithm, it will never be the case. Even in the case of other confident algorithms, it is not always the case, since the other confident algorithm can choose to cooperate with another player, or not to cooperate.

Second, to address the shortcomings of the Confident algorithm, we devised the Symmetric Confident algorithm. It does run the confident algorithm for each player. So for each player we have the mean result of all coalitions with all players. We can then find for each player the best coalition. If the best coalition for the player to move is also the best coalition for the other player of the coalition, then the player to move chooses the move of the UCT tree of the coalition. If this is not the case, the player chooses the move of a coalition that is better than no coalition if the other player of the coalition has its best coalition with the player. In all other cases, it chooses the move of the UCT tree without coalition.

The third confident algorithm is the Same algorithm. In contrast to the two previous algorithms, it assumes that it knows who the other Same algorithms are. It consists in choosing the best coalition among the other Same players, including itself (choosing itself is equivalent to no coalition).

5 Experimental Results

The experiments consist in playing the algorithms against each other. All our results use 200 9×9 games experiments.

5.1 UCT

The *Surrounded* algorithm always avoids playing on an intersection which has all its adjacent intersections of the same color as the player to move. It is a simple rule used in the playouts. The *Eye* algorithm also verifies the status of the diagonals of the intersection in order to avoid playing on virtual eyes during playouts.

Table 1 provides the results of three-player games. Each line gives the mean number of points of each algorithm playing 200 9×9 games against the other algorithms.

Given the first line that only contains Surrounded algorithms, and the lines that only contain the RAVE algorithm (lines 7 and 8), we can see that the value of komi is low in three-player Go (approximately one point for White and two points for Red given the RAVE results for 10,000 playouts). This is our first surprising result, we expected the third player to be more at a disadvantage than only two points.

Table 1. Results for different UCT algorithms

	Playouts	Black	Black points	White	White points	Red	Red points
1	1,000	Surrounded	29.78	Surrounded	25.02	Surrounded	26.20
2	1,000	Eye	26.80	Surrounded	27.76	Surrounded	26.42
3	1,000	Surrounded	27.91	Eye	28.66	Eye	24.44
4	1,000	Pattern	24.46	Surrounded	33.31	Surrounded	23.23
5	1,000	Surrounded	23.63	Pattern	23.82	Surrounded	33.56
6	1,000	RAVE	36.40	Surrounded	21.34	Surrounded	23.26
7	1,000	RAVE	27.94	RAVE	26.74	RAVE	26.32
8	10,000	RAVE	28.27	RAVE	27.13	RAVE	25.60

The first three lines of Table 1 compare (1) algorithms that use surrounded intersections as the avoiding rule, and (2) algorithms that use real eyes as the avoiding rule. In two-player Go, using virtual eyes is superior to using surrounded intersections [3]. Surprisingly, it is not the case in three-player Go as can be seen from the first three lines of the table. I feel uneasy giving explanations about the strength of playout policies as it is a very non-intuitive topic, it has been observed that a *better* playout player (i.e., one that often wins playouts against a simple one) can give worse results than straightforward playout players when used in combination with tree search. In all the following algorithms we have used the *Surrounded* rule.

The fourth and fifth lines of Table 1 test an algorithm using MOGO patterns in the playouts [7]. It uses exactly the same policy as described in the report. Friend stones in the patterns are matched to friend stones on the board, and enemy stones in the patterns can be matched with all colors different from the color of friend stones on the board. In contrast to two-player Go, MOGO's patterns give worse results than no pattern, and tend to favor the player following the player using patterns. A possible explanation for the bad results of patterns may be that in multi-player Go moves are less often close to the previous move than in usual Go. The algorithms of Subsections 5.2 to 5.4 do not use patterns in the playouts.

The lines 6 to 8 of Table 1 test the *RAVE* algorithm. When matched again two *Surrounded* UCT-based algorithms, it scores 36.40 points which is much better than standard UCT. When three RAVE algorithms are used, the results come close to the results of the standard UCT algorithms. All the following algorithms (of Subsections 5.2 to 5.4) use the RAVE optimization.

5.2 Paranoid UCT

Table 2 gives the results of the Paranoid algorithm matched against the RAVE algorithm. Paranoid against two RAVE algorithms has better results than both of them. Two Paranoid algorithms against one RAVE algorithm have also better results. When three Paranoid algorithms are matched, the scores come back to the equilibrium. So in these experiments, Paranoid is better than RAVE.

Table 2. Results for the Paranoid algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Paranoid	31.71	RAVE	26.89	RAVE	22.38
1,000	Paranoid	29.82	Paranoid	28.48	RAVE	22.67
1,000	Paranoid	29.11	Paranoid	26.85	Paranoid	25.04

5.3 Alliance UCT

Table 3 gives the results of the Alliance algorithm with different scoring systems for the playouts. When using Normal scoring, Alliance is not much beneficial against Paranoid, but has better results against RAVE.

Using the Allies' scoring gives better results for the Alliance and much worse results for Paranoid and RAVE. The best results for the Alliance are obtained with Joint scoring, in this case Paranoid only scores 1.57 on average and RAVE 0.30.

Table 3. Results for the Alliance algorithms

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Alliance(Normal)	28.69	Alliance(Normal)	25.09	Paranoid	24.58
1,000	Alliance(Normal)	32.15	Alliance(Normal)	30.44	RAVE	14.56
1,000	Alliance(Allies)	30.78	Alliance(Allies)	29.85	Paranoid	6.91
1,000	Alliance(Allies)	32.88	Alliance(Allies)	30.55	RAVE	2.64
1,000	Alliance(Joint)	34.38	Alliance(Joint)	33.12	Paranoid	1.57
1,000	Alliance(Joint)	35.61	Alliance(Joint)	34.01	RAVE	0.30

5.4 Confident UCT

Below we address the various confident algorithms. In our tests, we used 1,000 playouts for each tree developed by the various confident algorithms. Table 4 gives the results of the Confident algorithms against the RAVE and the Paranoid algorithms. The Paranoid and the RAVE algorithms are clearly better than the confident algorithm, and Paranoid is slightly better than RAVE.

Table 4. Results for the Confident algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Confid(Normal)	18.08	Confid(Normal)	23.84	Paranoid	39.08
1,000	Confid(Normal)	17.75	Confid(Normal)	24.86	RAVE	38.40
1,000	Confid(Allies)	14.22	Confid(Allies)	17.67	Paranoid	48.11
1,000	Confid(Allies)	10.99	Confid(Allies)	22.24	RAVE	45.97

Table 5 gives the results of the Symmetric Confident algorithms against the RAVE and the Paranoid algorithms. Here again, Paranoid and RAVE are better than Symmetric Confident, and Paranoid is slightly better than RAVE.

Table 5. Results for the Symmetric Confident algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Sym(Normal)	19.94	Sym(Normal)	16.67	Paranoid	42.97
1,000	Sym(Normal)	18.97	Sym(Normal)	19.54	RAVE	39.85

Table 6. Results for the Same algorithm

Playouts	Black	Black points	White	White points	Red	Red points
1,000	Same(Normal)	25.38	Same(Normal)	32.21	RAVE	23.34
1,000	Same(Normal)	25.61	Same(Normal)	27.59	Paranoid	27.79
1,000	Same(Allies)	32.53	Same(Allies)	37.76	RAVE	4.73
1,000	Same(Allies)	30.83	Same(Allies)	34.03	Paranoid	10.28
1,000	Same(Allies)	35.78	Same(Allies)	22.74	Same(Allies)	17.92
1,000	Same(Joint)	35.32	Same(Joint)	34.32	RAVE	0.58
1,000	Same(Joint)	33.62	Same(Joint)	34.06	Paranoid	1.70
1,000	Same(Allies)	64.96	Same(Joint)	7.95	Paranoid	4.76

In Table 6 two types of Same algorithms are tested against the Paranoid and RAVE algorithms. When the Same algorithm uses Normal scoring at the end of the playouts, playing two Same algorithms against a paranoid algorithm is not much more beneficial and even slightly worse than using paranoid algorithms as can be seen by comparing the results of Table 6 with those of Table 2. So, cooperation with Normal playout scoring is not much beneficial. However, when the playouts are scored according to the Allies' scoring, cooperation becomes much more interesting and the paranoid algorithm obtains much worse results (10.28 instead of 27.79). The RAVE algorithm is in this case even worse with only 4.73 points. For all these results, the games were scored with the Normal scoring algorithm. Only the playouts can be scored with the Allies' scoring algorithm. This is why the games played by the Same(Allies) algorithms do not sum up to 81 points, the Allies' algorithm stops playing when empty intersections are surrounded by allies, but the game is scored with the normal scoring and the empty intersections that are surrounded by more than one color are not counted.

When three Same(Allies) algorithms play together, the results favor the first player.

The next two lines of Table 6 give the results of the Same algorithm with Joint scoring of the playouts. The Same algorithm gives even better results in this case, and Paranoid and RAVE often end the games with no points at all.

When Same(Allies) plays against/with Same(Joint) and Paranoid, it scores extremely well. Paranoid is beaten when the two Same algorithms decide to make an alliance, and Same(Allies) beats Same(Joint) since it continues to make territory for itself at the end of the game when Same(Joint) considers it common territory.

5.5 Techniques Used in the Algorithms

Table 7 gives an overview of the techniques used for each algorithm.

Table 7. Techniques used for each algorithm

Algorithm	Playout policy	Search	Scoring function
Surrounded	Surrounded by player	UCT	stones + eyes
Eye	Eye by player	UCT	stones + eyes
Pattern	Surrounded by player + Patterns	UCT	stones + eyes
RAVE	Surrounded by player	RAVE	stones + eyes
Paranoid	Surrounded by coalition	RAVE	stones + eyes
Alliance(Normal)	Surrounded by coalition	RAVE	stones + eyes
Alliance(Allies)	Surrounded by coalition	RAVE	stones + common eyes
Alliance(Joint)	Surrounded by coalition	RAVE	common stones + common eyes
Confid(Normal)	Surrounded by coalition	RAVE	stones + eyes
Confid(Allies)	Surrounded by coalition	RAVE	stones + common eyes
Sym(Normal)	Surrounded by coalition	RAVE	stones + eyes
Same(Normal)	Surrounded by coalition	RAVE	stones + eyes
Same(Allies)	Surrounded by coalition	RAVE	stones + common eyes
Same(Joint)	Surrounded by coalition	RAVE	common stones + common eyes

6 A Summary of Findings

We addressed the application of UCT to multi-player games, and more specifically to multi-player Go. We defined a simple and effective heuristic, used in the playouts, that models coalitions of players. This heuristic has been used in the Alliance algorithm and can be also effective when used with the appropriate scoring of the playouts. The Paranoid algorithm, which assumes a coalition of the other players plays better than the usual RAVE algorithm. Confident algorithms, such as Confident and Symmetric Confident, that choose to cooperate when it is most beneficial are worse than the Paranoid algorithm when they are not aware of the other players algorithms. However, when the players' algorithms are known, as in the Same algorithm, they become better than the Paranoid algorithm.

If a multi-player Go tournament were organized, the best algorithm would be dependent on the available information on the competitors. Future work will address the extension of coalition algorithms to more than three players, the effects of communications between players as well as the parallelization of the algorithms [4].

References

1. Bouzy, B., Cazenave, T.: Computer Go: An AI-Oriented Survey. Artificial Intelligence 132(1), 39–103 (2001)
2. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)

3. Cazenave, T.: Playing the right atari. ICGA Journal 30(1), 35–42 (2007)
4. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Computer Games Workshop 2007, Amsterdam, The Netherlands, June 2007, pp. 93–101 (2007)
5. Coulom, R.: Efficient selectivity and back-up 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. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML, pp. 273–280 (2007)
7. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in monte-carlo go. Technical Report 6062, INRIA (2006)
8. 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)
9. Korf, R.E.: Multi-player alpha-beta pruning. Artif. Intell. 48(1), 99–111 (1991)
10. Loeb, D.E.: Stable winning coalitions. In: Games of No Chance, vol. 29, pp. 451–471 (1996)
11. Luckhart, C., Irani, K.B.: An algorithmic solution of n-person games. In: AAAI, pp. 158–162 (1986)
12. Sturtevant, N.R.: Last-branch and speculative pruning algorithms for $\max^n$. In: IJCAI, pp. 669–678 (2003)
13. Sturtevant, N.R.: Current challenges in multi-player game search. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 285–300. Springer, Heidelberg (2006)
14. Sturtevant, N.R., Korf, R.E.: On pruning techniques for multi-player games. In: AAAI/IAAI, pp. 201–207 (2000)

Parallel Monte-Carlo Tree Search

Guillaume M.J.-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik

Games and AI Group, MICC, Faculty of Humanities and Sciences,
Universiteit Maastricht, Maastricht, The Netherlands
{g.chaslot,m.winands,herik}@micc.unimaas.nl

Abstract. Monte-Carlo Tree Search (MCTS) is a new best-first search method that started a revolution in the field of Computer Go. Parallelizing MCTS is an important way to increase the strength of any Go program. In this article, we discuss three parallelization methods for MCTS: *leaf parallelization*, *root parallelization*, and *tree parallelization*. To be effective tree parallelization requires two techniques: adequately handling of (1) *local mutexes* and (2) *virtual loss*. Experiments in 13×13 Go reveal that in the program MANGO root parallelization may lead to the best results for a specific time setting and specific program parameters. However, as soon as the selection mechanism is able to handle more adequately the balance of exploitation and exploration, tree parallelization should have attention too and could become a second choice for parallelizing MCTS. Preliminary experiments on the smaller 9×9 board provide promising prospects for tree parallelization.

1 Introduction

For decades, the standard method for two-player games such as chess and checkers has been $\alpha\beta$ search. Nevertheless, in 2006 Monte-Carlo Tree Search (MCTS) [4, 6, 7, 8, 10, 12] started a revolution in the field of Computer Go. At this moment (January 2008) the best MCTS 9×9 Go programs are ranked 500 rating points higher than the traditional programs on the Computer Go Server [2]. On the 19×19 Go board, MCTS programs are also amongst the best programs. For instance, the MCTS program MOGO won the Computer Olympiad 2007 [9], and the MCTS program CRAZY STONE has the highest rating amongst programs on the KGS Go Server [1].

MCTS is not a classical tree search followed by a Monte-Carlo evaluation, but rather a best-first search guided by the results of Monte-Carlo simulations. Just as for $\alpha\beta$ search [11], it holds for MCTS that the more time is spent for selecting a move, the better the game play is. Moreover, the law of diminishing returns that nowadays has come into effect for many $\alpha\beta$ chess programs, appears to be less of an issue for MCTS Go programs. Hence, parallelizing MCTS seems to be a promising way to increase the strength of a Go program. Pioneer work has been done by Cazenave and Jouandeau [3] by experimenting with two parallelization methods: leaf parallelization and root parallelization (original called single-run parallelization).

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 60–71, 2008.

In this article we introduce a third parallelization method, called tree parallelization. We compare the three parallelization methods (leaf, root, and tree) by using the *Games-Per-Second (GPS) speedup measure* and *strength-speedup measure*. The first measure corresponds to the improvement in speed, and the second measure corresponds to the improvement in playing strength. The three parallelization methods are implemented and tested in our Go program MANGO [5] (mainly designed and constructed by Guillaume Chaslot), running on a multi-core machine containing 16 cores. An earlier version of the program - using more modest hardware - participated in numerous international competitions in 2007, on board sizes 13×13 and 19×19. It was ranked in the first half of the participants at all these events [1, 9].

The article is organized as follows. In Sect. 2 we present the basic structure of MCTS. In Sect. 3, we discuss the different methods used to parallelize an MCTS program. We empirically evaluate the three parallelization methods in Sect. 4. Section 5 provides the conclusions and describes future research.

2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [7, 12] is a best-first search method that does not require a positional evaluation function. It is based on randomized explorations of the search space. Using the results of previous explorations, the algorithm gradually grows a game tree in memory, and successively becomes better at accurately estimating the values of the most promising moves.

MCTS consists of four strategic phases, repeated as long as there is time left. The phases are as follows. (1) In the *selection step* the tree is traversed from the root node until it selects a leaf node L that is not added to the tree yet.[1] (2) Subsequently, the *expansion strategy* is called to add the leaf node L to the tree. (3) A *simulation strategy* plays moves in a self-play mode until the end of the

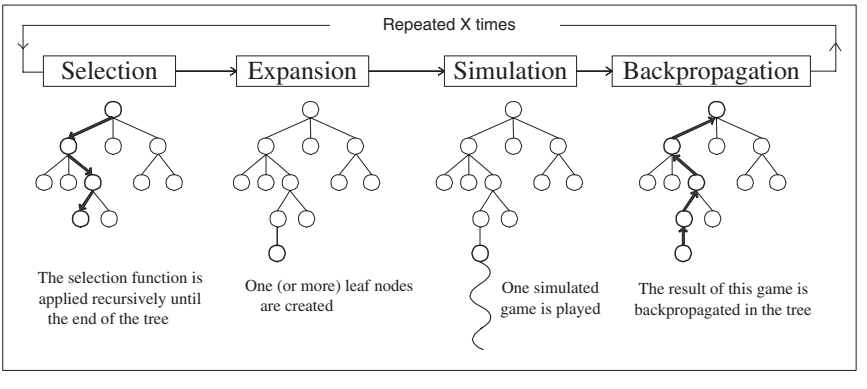


Fig. 1. Scheme of Monte-Carlo Tree Search

[1] Examples of such a strategy are UCT, OMC, BAST, etc. [4, 6, 7, 12]. All experiments have been performed with the UCT algorithm [12] using a coefficient C_p of 0.35.

game is reached. The result R of such a "simulated" game is $+1$ in case of a win for Black (the first player in Go), 0 in case of a draw, and -1 in case of a win for White. (4) A *backpropagation strategy* propagates the results R through the tree, i.e., in each node traversed the average result of the simulations is computed. The four phases of MCTS are shown in Fig. 1.

When all the time is consumed, the move played by the program is the root child with the highest visit count. It might be noticed that MCTS can be stopped anytime. However, the longer the program runs, the stronger the program plays. We show in Sect. 4 that the rating of our program increases nearly linearly with the logarithm of the time spent.

3 Parallelization of Monte-Carlo Tree Search

In this article, we consider parallelization for a symmetric multiprocessor (SMP) computer. We always use one processor thread for each processor core. One of the properties of a SMP computer is that any thread can access the central (shared) memory with the same (generally low) latency. As a consequence, parallel threads should use a mutual exclusion (mutex) mechanism in order to prevent any data corruption, due to simultaneous memory access. This could happen when several threads are accessing the MCTS tree (i.e., in phase 1, 2 or 4). However, the simulation phase (i.e., phase 3) does not require any information from the tree. There, simulated games can be played completely independently from each other. This specific property of MCTS is particularly interesting for the parallelization process. For instance, it implies that long simulated games make the parallelization easier. We distinguish three main types of parallelization, depending on which phase of the Monte-Carlo Tree Search is parallelized: leaf parallelization, root parallelization, and tree parallelization.

3.1 Leaf Parallelization

Leaf parallelization introduced by Cazenave and Jouandeau [3] is one of the easiest ways to parallelize MCTS. To formulate it in machine-dependent terms, only one thread traverses the tree and adds one of more nodes to the tree when a leaf node is reached (phase 1 and 2). Next, starting from the *leaf* node, independent simulated games are played for each available thread (phase 3). When all games are finished, the result of all these simulated games is propagated backwards through the tree by one single thread (phase 4). Leaf parallelization is depicted in Fig. 2a.

Leaf parallelization seems interesting because its implementation is easy and does not require any mutexes. However, two problems appear. First, the time required for a simulated game is highly unpredictable. Playing n games using n different threads takes more time in average than playing one single game using one thread, since the program needs to wait for the longest simulated game. Second, information is not shared. For instance, if 16 threads are available, and 8 (faster) finished games are all losses, it will be highly probable that most games

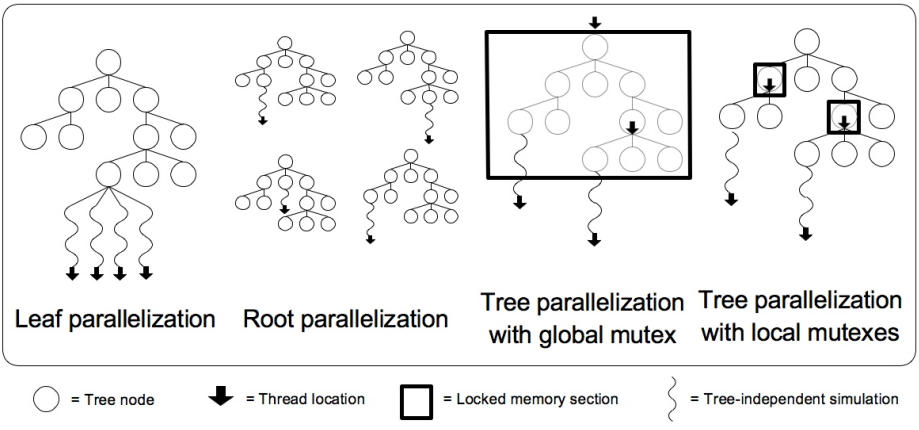


Fig. 2. (a) Leaf parallelization (b) Root parallelization (c) Tree parallelization with global mutex (d) and with local mutexes

will lead to a loss. Therefore, playing 8 more games is a waste of computational power. To decrease the waiting time, the program might stop the simulations that are still running when the results of the finished simulations become available. This strategy would enable the program to traverse the tree more often, but some threads would be idle. Leaf parallelization can be performed inside an SMP environment, or even on a cluster using MPI (Message Passing Interface) communication.

3.2 Root Parallelization

Cazenave proposed a second parallelization under the name "single-run" parallelization [3]. In this article we will call it *root parallelization* to stress the part of the tree for which it applies. The method works as follows. It consists of building multiple MCTS trees in parallel, with one thread per tree. Similar to leaf parallelization, the threads do not share information with each other. When the available time is spent, all the root children of the separate MCTS trees are merged with their corresponding clones. For each group of clones, the scores of all games played are added. The best move is selected based on this grand total. This parallelization method only requires a minimal amount of communication between threads, so the parallelization is easy, even on a cluster. Root parallelization is depicted in Fig. 2b.

3.3 Tree Parallelization

In this article we introduce a new parallelization method called *tree parallelization*. This method uses one shared tree from which several simultaneous games are played. Each thread can modify the information contained in the tree; therefore mutexes are used to lock from time to time certain parts of the tree to

prevent data corruption. There are two methods to improve the performance of tree parallelization: (1) mutex location and (2) "virtual loss".

Mutex location. Based on the location of the mutexes in the tree, we distinguish two mutex location methods: (1) using a *global mutex* and (2) using several *local mutexes*.

The global mutex method locks the whole tree in such a way that only *one* thread can access the search tree at a time (phase 1, 2, and 4). In the meantime several other processes can play simulated games (phase 3) starting from *different* leaf nodes. This is a major difference with leaf parallelization where all simulated games start from the *same* leaf node. The global mutex method is depicted in Fig. 2c. The potential speedup given by the parallelization is bounded by the time that has to be spent in the tree. Let x be the average percentage of time spent in the tree by one single thread. The maximum speedup in terms of games per second is $100/x$. In most MCTS programs x is relatively high (say between 25 to 50%), limiting the maximum speedup substantially. This is the main disadvantage of this method.

The local mutexes method makes it possible that *several* threads can access the search tree simultaneously. To prevent data corruption because two (or more) threads access the same node, we lock a node by using a local mutex when it is visited by a thread. At the moment a thread departs the node, it is unlocked. Thus, this solution requires to frequently lock and unlock parts of the tree. Hence, fast-access mutexes such as spinlocks have to be used to increase the maximum speedup. The local mutexes method is depicted in Fig. 2d.

Virtual loss. If several threads start from the root at the same time, it is possible that they traverse the tree for a large part in the same way. Simulated games might start from leaf nodes, which are in the neighborhood of each other. It can even happen that simulated games begin from the same leaf node. Because a search tree typically has millions of nodes, it may be redundant to explore a rather small part of the tree several times. Coulom[2] suggests to assign one "virtual loss" when a node is visited by a thread (i.e., in phase 1). Hence, the value of this node will be decreased. The next thread will only select the same node if its value remains better than its siblings' values. The virtual loss is removed when the thread that gave the virtual loss starts propagating the result of the finished simulated game (i.e., in phase 4). Owing to this mechanism, nodes that are clearly better than others will still be explored by all threads, while nodes for which the value is uncertain will not be explored by more than one thread. Hence, this method keeps a certain balance between exploration and exploitation in a parallelized MCTS program.

4 Experiments

In this section we compare the different parallelization methods with each other. Subsection 4.1 discusses the experimental set-up. We show the performance of

[2] Personal Communication.

leaf parallelization, root parallelization, and tree parallelization in Subsection 4.2, 4.3, and 4.4, respectively. An overview of the results is given in Subsection 4.5. Root parallelization and tree parallelization are compared under different conditions in Subsection 4.6.

4.1 Experimental Set-Up

The aim of the experiments is to measure the quality of the parallelization process. We use two measures to evaluate the speedup given by the different parallelization methods. The first measure is called the *Games-Per-Second (GPS) speedup*. It is computed by dividing the number of simulated games per second performed by the multithreaded program, by the number of games per second played by a single-threaded program. However, the GPS speedup measure might be misleading, since it is not always the case that a faster program is stronger. Therefore, we propose a second measure: called *strength-speedup*. It corresponds to the *increase of time needed to achieve the same strength*. For instance, a multithreaded program with a strength-speedup of 8.5 has the same strength as a single-threaded program, which consumes 8.5 times more time.

In order to design the strength-speedup measurement, we proceed in three steps. First, we measure the strength of our Go program MANGO on the 13×13 board against GNU GO 3.7.10, level 0, for 1 second, 2 seconds, 4 seconds, 8 seconds, and 16 seconds. For each time setting, 2,000 games are played. Figure 3 reports the strength of MANGO in terms of percentage of victory. In Fig. 4, the increase in strength in term of rating points as a function of the logarithmic time is shown. This function can be approximated accurately by linear regression, using a correlation coefficient $R^2 = 0.9922$. Second, the linear approximation is used to give a theoretical Go rating for any amount of time. Let us assume

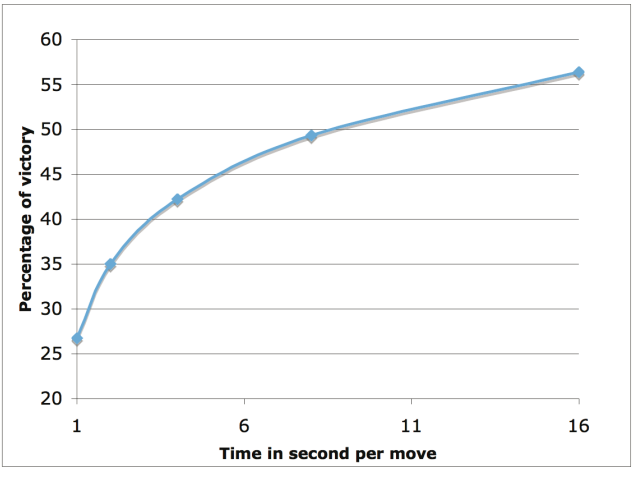


Fig. 3. Scalability of the strength of MANGO with time

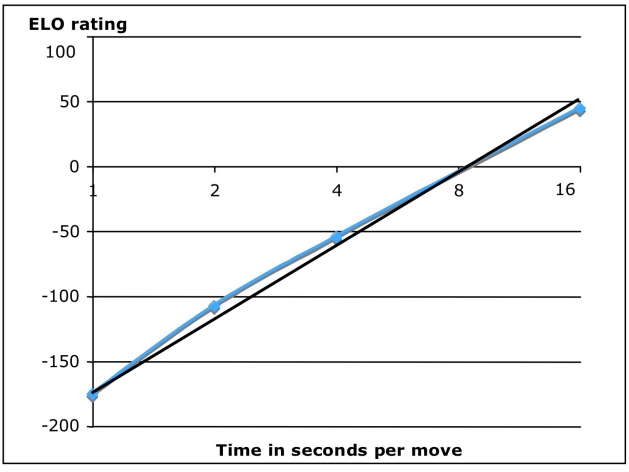


Fig. 4. Scalability of the rating of MANGO vs. GNU GO with time. The curve represents the data points, and the line is a trend-line for this data.

that E_t is the level of the program in rating points, T is the time in seconds per move. Linear regression gives us $E_t(T) = A \cdot log_2 T + B$ with $A = 56.7$ and $B = -175.2$. Third, the level of play of the multithreaded program is measured against the same version of GNU GO, with one second per move. Let E_m be the rating of this program against GNU GO. The strength-speedup S is defined by: $S \in \mathbb{R} | E_t(S) = E_m$.

The experiments were performed on the supercomputer Huygens, which has 120 nodes, each with 16 cores POWER5 running at 1.9 GHz and having 64 Gigabytes of memory per node. Using this hardware the single-threaded version of MANGO was able to perform 3,400 games per second in the initial board position of 13×13 Go. The time setting used for the multithreaded program was 1 second per move.

4.2 Leaf Parallelization

In the first series of experiments we tested the performance of plain leaf parallelization. We did not use any kind enhancement to improve this parallelization method as discussed in Subsection 3.1. The results regarding winning percentage, GPS speedup, and strength-speedup for 1, 2, 4, and 16 threads are given in Table 1. We observed that the GPS speedup is quite low. For instance, when running 4 simulated games in parallel, finishing all of them took 1.15 times longer than finishing just 1 simulated game. For 16 threads, it took two times longer to finish all games compared by finishing just one. The results show that the strength-speedup obtained is rather low as well (2.4 for 16 processors). So, we may conclude that plain leaf parallelization is not a good way for parallelizing MCTS.

Table 1. Leaf parallelization

Number of threads	Winning percentage	Number of games	Confidence interval	GPS Speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	26.8 %	2000	2.0 %	1.8	1.2
4	32.0 %	1000	2.8 %	3.3	1.7
16	36.5 %	500	4.3 %	7.6	2.4

4.3 Root Parallelization

In the second series of experiments we tested the performance of root parallelization. The results regarding winning percentage, GPS speedup, and strength-speedup for 1, 2, 4, and 16 threads are given in Table 2.

Table 2 indicates that root parallelization is a quite effective way of paralleling MCTS. One particularly interesting result is that, for four processor threads, the strength-speedup is significantly higher than the number of threads used (6.5 instead of 4). This result implies that, in our program MANGO, it is more efficient to run four independent MCTS searches of one second than to run one large MCTS search of four seconds. It might be that the algorithm stays for quite a long time in local optima. This effect is caused by the UCT coefficient setting. For small UCT coefficients, the UCT algorithm is able to search more deeply in the tree, but also stays a longer time in local optima. For high coefficients, the algorithm escapes more easily from the local optima, but the resulting search is shallower. The optimal coefficient for a specific position can only be determined experimentally. The time setting also influences the scalability of the results. For a short time setting, the algorithm is more likely to spend too much time in local optima. Hence, we believe that with higher time settings, root parallelization will be less efficient. In any case, we may conclude that root parallelization is a simple and effective way to parallelize MCTS.

Table 2. Root parallelization

Number of threads	Winning Percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1	1.0
2	38.0 %	2000	2.2 %	2	3.0
4	46.8 %	2000	2.2 %	4	6.5
16	56.5 %	2000	2.2 %	16	14.9

4.4 Tree Parallelization

In the third series of experiments we tested the performance of tree parallelization. Below, we have a closer look at the *mutexes location* and *virtual loss*.

Table 3. Tree parallelization with global mutex

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	31.3 %	2000	2.2 %	1.8	1.6
4	37.9 %	2000	2.2 %	3.2	3.0
16	36.5 %	500	4.5 %	4.0	2.6

Table 4. Tree parallelization with local mutex

Number of threads	Percentage of victory	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	32.9 %	2000	2.2 %	1.9	1.9
4	38.4 %	2000	2.2 %	3.6	3.0
16	39.9 %	500	4.4 %	8.0	3.3

Table 5. Using virtual loss for tree parallelization

Number of threads	Winning percentage	Number of games	Confidence interval	GPS speedup	Strength speedup
1	26.7 %	2000	2.2 %	1.0	1.0
2	33.8 %	2000	2.2 %	1.9	2.0
4	40.2 %	2000	2.2 %	3.6	3.6
16	49.9 %	2000	2.2 %	9.1	8.5

Mutexes location. First, the global mutex method was tested. The results are given in Table 3. These results show that the strength-speedup obtained up to 4 threads is satisfactory (i.e., strength-speedup is 3). However, for 16 threads, this method is clearly insufficient. The strength-speedup drops from 3 for 4 threads to 2.6 for 16 threads. So, we may conclude that the global mutex method should not be used in tree parallelization.

Next, we tested the performance for the local mutexes method. The results are given in Table 4. Table 4 shows that for each number of threads the local mutexes has a better strength-speed than global mutex. Moreover, by using local mutexes instead of global mutex the number of games played per second is doubled when using 16 processor threads. However, the strength-speedup for 16 processors threads is just 3.3. Compared to the result of root parallelization (14.9 for 16 threads), this result is quite disappointing.

Using virtual loss. Based on the previous results we extended the global mutexes tree parallelization with the virtual loss enhancement. The results of using virtual loss are given in Table 5.

Table 5 shows that the effect of the virtual loss when using 4 processor threads is moderate. If we compare the strength-speedup of Table 4 we see an increase

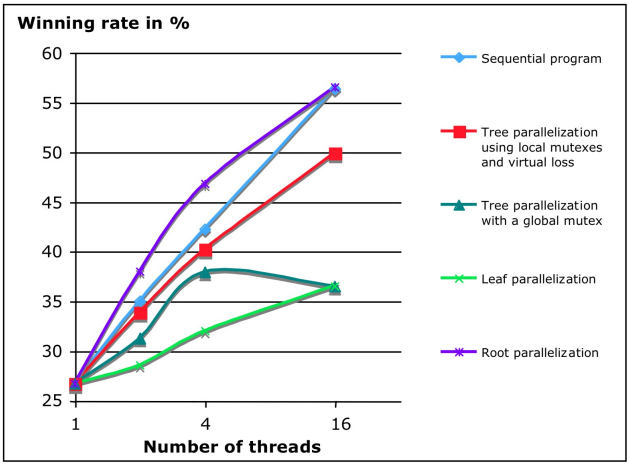


Fig. 5. Performance of the different parallelization methods

from 3.0 to 3.6. But when using 16 processor threads, the result is more impressive. Tree parallelization with virtual loss is able to win 49.9% of the games instead of 39.9% when it is not used. The strength-speedup of tree parallelization increases from 3.3 (see Table 4) to 8.5. Thus, we may conclude that virtual loss is important for the performance of tree parallelization when the number of processor threads is high.

4.5 Overview

In Fig. 5 we have depicted the performance of leaf parallelization, root parallelization, and tree parallelization with global mutex or with local mutexes. The x-axis represents the logarithmic number of threads used. The y-axis represents the winning percentage against GNU Go. For comparison reasons, we have plotted the performance of the default (sequential) program when given more time instead of more processing power. We see that root parallelization is superior to all other parallelization methods, performing even better than the sequential program.

4.6 Root Parallelization vs. Tree Parallelization Revisited

In the previous subsection we saw that on the 13×13 board root parallelization outperformed all other parallelization methods, including tree parallelization. It appears that the strength of root parallelization lies not only in an more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. The results could be different for other board sizes, time settings, and parameter settings. Therefore, we switched to a different board size (9×9) and three different time settings (0.25, 2.5, and 10 seconds per move). Using 4

Table 6. 9×9 results for root and tree parallelization using 4 threads

Time (s)	Winning percentage	
	Root parallelization	Tree parallelization
0.25	60.2 %	63.9 %
2.50	78.7 %	79.3 %
10.0	87.2 %	89.2 %

processor threads, root, and tree parallelization played both 250 games against the same version of GNU Go for each time setting. The results are given in Table 6. For 4 threads, we see that root parallelization and tree parallelization perform equally well now. Nevertheless, the number of games played and the number of threads used is not sufficient to give a definite answer which method is better.

5 Conclusions and Future Research

In this article we discussed the use of leaf parallelization and root parallelization for parallelizing MCTS. We introduced a new parallelization method, called tree parallelization. This method uses one shared tree from which games simultaneously are played. Experiments were performed to assess the performance of the parallelization methods in the Go program Mango on the 13×13 board. In order to evaluate the experiments, we propose the strength-speedup measure, which corresponds to the time needed to achieve the same strength. Experimental results indicated that leaf parallelization was the weakest parallelization method. The method led to a strength-speedup of 2.4 for 16 processor threads. The simple root parallelization turned out to be the best way for parallelizing MCTS. The method led to a strength-speedup of 14.9 for 16 processor threads. We saw that tree parallelization requires two techniques to be effective. First, using local mutexes instead of global mutex doubles the number of games played per second. Second, virtual loss increases both the speed and the strength of the program significantly. By using these two techniques, we obtained a strength-speedup of 8.5 for 16 processor threads.

Despite the fact that tree parallelization is still behind root parallelization, it is too early to conclude that root parallelization is the best way of parallelization. It transpires that the strength of root parallelization lies not only in an more effective way of parallelizing MCTS, but also in preventing that MCTS stays too long in local optima. Root parallelization repairs (partially) a problem in the UCT formula used by the selection mechanism, namely handling the issue of balancing exploitation and exploration. For now, we may conclude that root parallelization lead to excellent results for a specific time setting and specific program parameters. However, as soon as the selection mechanism is able to handle more adequately the balance of exploitation and exploration, we believe that tree parallelization could become the choice for parallelizing MCTS.

Preliminary experiments on the smaller 9×9 board suggest that tree parallelization is at least as strong as root parallelization.

In this paper we limited the tree parallelization to one SMP-node. In future research, we will focus on tree parallelization and determine under which circumstances tree parallelization outperforms root parallelization. We believe that the selection strategy, the time setting, and the board size are important factors. Finally, we will test tree parallelization for a cluster with several SMP-nodes.

Acknowledgments. The authors thank Bruno Bouzy for providing valuable comments on an early draft of this paper. This work is financed by the Dutch Organization for Scientific Research (NWO) for the project Go for Go, grant number 612.066.409. The experiments were run on the supercomputer Huygens provided by the Nationale Computer Faciliteiten (NCF).

References

1. KGS Go Server Tournaments, http://www.weddslist.com/kgs/past/index.html
2. Computer Go Server (2008), http://cgos.boardspace.net
3. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: van den Herik, H.J., Uiterwijk, J.W.H.M., Winands, M.H.M., Schadd, M.P.D. (eds.) Proceedings of the Computer Games Workshop 2007 (CGW 2007), The Netherlands, pp. 93–101. Universiteit Maastricht, Maastricht (2007)
4. Chaslot, G.M.J.-B., Saito, J.-T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo Strategies for Computer Go. In: Schobbens, P.-Y., Vanhoof, W., Schwanen, G. (eds.) Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, pp. 83–90 (2006)
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. Coquelin, P.-A., Munos, R.: Bandit algorithms for tree search. In: proceedings of Uncertainty in Artificial Intelligence, Vancouver, Canada (to appear, 2007)
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. Gelly, S., Wang, Y.: Exploration Exploitation in Go: UCT for Monte-Carlo Go. In: Twentieth Annual Conference on Neural Information Processing Systems (NIPS 2006) (2006)
9. Gelly, S., Wang, Y.: Mogo wins 19×19 go tournament. ICGA Journal 30(2), 111–112 (2007)
10. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
11. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)
12. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)

A Parallel Monte-Carlo Tree Search Algorithm

Tristan Cazenave and Nicolas Jouandeau

LIASD, Université Paris 8, 93526, Saint-Denis, France
{cazenave,n}@ai.univ-paris8.fr

Abstract. Monte-Carlo Tree Search is a powerful paradigm for the game of Go. In this contribution we present a parallel Master-Slave algorithm for Monte-Carlo Tree Search and test it on a network of computers using various configurations: from 12,500 to 100,000 playouts, from 1 to 64 slaves, and from 1 to 16 computers. On our own architecture we obtain a speedup of 14 for 16 slaves. With a single slave and five seconds per move our algorithm scores 40.5% against GNU Go, with sixteen slaves and five seconds per move it scores 70.5%. At the end we give the potential speedups of our algorithm for various playout times.

1 Introduction

So far, contributions on parallelization in games are mostly about the parallelization of the Alpha-Beta algorithm [2]. Here we address the parallelization of the UCT algorithm (Upper Confidence bounds applied to Trees). This contribution is an improvement over our previous work on the parallelization of UCT [3], where we tested three different algorithms. We summarize them below. (1) The single-run algorithm uses not much communication, it consists in having each slave computing its own UCT tree independently of the others. When the thinking time is elapsed, it combines the results of the different slaves to choose its move. (2) The multiple-runs algorithm periodically updates the trees with the results of the other slaves. (3) The at-the-leaves algorithm computes multiple playouts in parallel at each leaf of the UCT tree.

In this paper we propose a different parallel algorithm that develops the UCT tree in the master part and performs the playouts in the slaves in parallel, it is close to the algorithm we presented orally at the Computer Games Workshop 2007 and that we used at the 2007 Computer Olympiad in Amsterdam. In passing we remark that Monte-Carlo Go has recently improved and is now able to compete with the best Go programs [4,6,7]. We show that it can be further improved using parallelization.

Section 2 describes related work. Section 3 presents the parallel algorithm. Section 4 provides experimental results. Section 5 gives a summary of findings.

2 Related Work

In this section we discuss some related work on Monte-Carlo Go. We first explain basic Monte-Carlo Go as implemented in Gobble in 1993. Then we address the combination of search and Monte-Carlo Go, followed by the UCT algorithm.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 72–80, 2008.

2.1 Monte-Carlo Go

The first Monte-Carlo Go program is GOBBLE [1]. It uses simulated annealing on a list of moves. The list is sorted by the mean score of the games where the move has been played. Moves in the list are switched with their neighbor with a probability dependent on the temperature. The moves are tried in the playouts in the order of the list. At the end, the temperature is set to zero for a small number of games. After all playouts have been played, the value of a move is the average score of the playouts in which the move has been played as first move. GOBBLE-like programs have a good global sense but lack tactical knowledge. For example, they often play useless Ataris, or try to save captured strings.

2.2 Search and Monte-Carlo Go

An effective way to combine search with Monte-Carlo Go has been found by Rémi Coulom and implemented in his program CRAZY STONE [4]. It consists in adding a leaf to the tree for each simulation. The choice of the move to develop in the tree depends on the comparison of (1) the results of the previous simulations that went through this node, with (2) the results of the simulations that went through its sibling nodes.

2.3 UCT

The UCT algorithm has been introduced recently [8], and it has been applied with success to Monte-Carlo Go in the program MOGO [6,7] among others.

When choosing a move to explore, there is a balance between exploitation (exploring the best move so far), and exploration (exploring other moves to see if they can be proved to be better). The UCT algorithm addresses the exploration/exploitation problem. UCT consists in exploring the move that maximizes $\mu_i + C \times \sqrt{\frac{log(games)}{child_i \rightarrow games}}$. The mean result of the games that start with the c_i move is μ_i, the number of games played in the current node is $games$, and the number of games that start with move c_i is $child_i \rightarrow games$.

The C constant can be used to adjust the level of exploration of the algorithm. High values favor exploration and low values favor exploitation.

3 Parallelization

In this section, we present the run-time environment used to execute processes on a cluster. Then we present and comment the master part of the parallel algorithm. Eventually, we present the slave part of the parallel algorithm.

3.1 The Parallel Run-Time Environment

To improve search, we choose message passing as parallel programming model, which is implemented in the standard MPI, also supported by Open MPI [5].

Open MPI is designed to achieve high performance computing on heterogeneous clusters. Our cluster is constituted with classical personal computers and with an SMP head node that has four processors. The resulting cluster is a private network connected with a TCP Gigabit network. Both communications are done only with the global communicator MPI_COMM_WORLD. Each hyper-threaded computer that allows to work on two threads at once, supports of one to four nodes of our parallel computer. Each node runs one task with independent data. Tasks are created at the beginning of the program's execution, via the use of the master-slave model. The SMP head node is always the master. All Go Text Protocol read and write commands are realized from and to the master. Slaves satisfy computing requests.

3.2 The Master Process

The master process is responsible for descending and updating the UCT tree. The slaves do the playouts that start with a sequence of moves sent by the master.

The master starts sending the position to each slave. Then it develops the UCT tree once for each slave and sends them an initial sequence of moves. Then it starts its main loop (called MasterLoop) which repeatedly receive from a slave the result of the playout starting with the sent sequence, update the UCT tree

```
1   Master ()
2       MasterLoop(board[ ], color, ko, time);
3       for(s ← 0; s < nbSlaves; s++)
4           | send(s, END_LOOP);
5       return bestUCTMove ();

6   MasterLoop(board[ ], color, ko, time)
7       for(s ← 0; s < nbSlaves; s++)
3           | send(s, board[ ], color, ko);
4           | seq[s][ ] ← descendUCTTree ();
5           | send(s, seq[s][ ]);
6       while(moreTime(time))
7           | s ←receive();
8           | result[s] ←receive();
9           | updateUCTTree(seq[s][ ], result[s]);
10          | seq[s][ ] ← descendUCTTree ();
11          | send(s, seq[s][ ]);
12      for(i ← 0; i < nbSlaves; i++)
13          | s ←receive();
14          | result[s] ←receive();
15          | updateUCTTree(seq[s][ ], result[s]);
```

Algorithm 1. Master Algorithm

with this result, create a new sequence descending the updated UCT tree, and sends this new sequence to the slave.

The master finishes the main loop when no more time is available or when the maximum number of playouts is reached. Before stopping, it receives the results from all the children that are still playing playouts until no more slave is active.

The master part of the parallel algorithm is given in Algorithm 1.

3.3 The Slave Process

The slave process loops until the master stops it with an END_GAME message, otherwise it receives the board, the color to play, and the ko intersection, and starts another loop in order to do playouts with this board configuration.

In this inner loop four actions are performed: (1) it starts receiving a sequence of moves, (2) it plays this sequence of moves on the board, (3) it completes a playout, and (4) it sends the result of the playout to the master process.

The slave part of the parallel algorithm is given in Algorithm 2.

```
1   SlaveLoop()
2       id ←slaveId()
3       while(true)
4           | if(receive(board[ ], color, ko) == END_GAME)
5           |     break;
6           | state ← CONTINUE;
7           | while(state == CONTINUE)
8           |     state ← SlavePlayout();
9       return;

10  SlavePlayout()
11      if(receive(sequence[ ]) == END_LOOP)
12          | return END_LOOP;
13      for(i ← 0; i < sequence.size(); i++)
14          | playMove(sequence[i]);
15      result ← playRandomGame();
16      send(id);
17      send(result);
18      return CONTINUE;
```

Algorithm 2. Slave Algorithm

4 Experimental Results

Tests are run on a simple network of computers running LINUX 2.6.18. The network includes 1 Gigabit switches, 16 computers with 1.86 GHz Intel dual core CPUs with 2 GB of RAM. The master process is run on the server which is a 3.20 GHz Intel Xeon with 4 GB of RAM.

Table 1. Results against GNU Go for 5 seconds per move

1 slave	40.50%
16 slaves	70.50%

Table 2. Results of the program against GNU Go 3.6

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
100,000 playouts	70.0%	69.0%	73.5%	70.0%	71.5%	65.0%	58.0%
50,000 playouts	63.5%	64.0%	65.0%	67.5%	65.5%	56.5%	51.5%
25,000 playouts	47.0%	49.5%	54.0%	56.0%	53.5%	48.5%	42.0%
12,500 playouts	47.5%	44.5%	44.0%	45.5%	45.0%	36.0%	32.0%

In our experiments, UCT uses $\mu_i + 0.3 \times \sqrt{\frac{log(games)}{child_i \rightarrow games}}$ to explore moves.

The random games are played using the same patterns as in Mogo [7] near the last move. If no pattern is matched near the last move, the selection of moves is the same as in Crazy Stone [4].

Table 1 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white, with komi 7.5) against GNU Go 3.6 default level. The time limit is set to five seconds per move. The first program uses one slave, it scores 40.5 % against GNU Go. The second program uses sixteen slaves, it scores 70.5 % against GNU Go.

Table 2 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white, with komi 7.5) for different numbers of slaves and different numbers of playouts of the parallel program against GNU Go 3.6 default level.

Table 2 can be used to evaluate the benefits from parallelizing the UCT algorithm. For example, in order to see if parallelizing on 8 slaves is more interesting than parallelizing with 4 slaves, we can compare the results of 100,000 playouts with 8 slaves (70.0%) to the results of 50,000 playouts with 4 slaves (65.0%). In this case, parallelization is beneficial since it gains 5.0% of wins against GNU Go 3.6. We compare 8 slaves with 100,000 playouts with 4 slaves with 50,000 playouts since they have close execution times (see Table 3).

To determine the gain of parallelizing with 8 slaves over not parallelizing at all, we compare the results of 12,500 playouts with 1 slave (47.5%) to the results of 100,000 playouts with 8 slaves (70.0%). In this case, the results are quite convincing. So, our conclusion is that parallelizing is beneficial.

A second interesting conclusion we may draw from the table is that the gain of parallelizing starts to decrease at 32 slaves. For example 100,000 playouts with 32 slaves wins 65.0% when 50,000 playouts with 16 slaves wins 65.5%. So, going from 16 slaves to 32 slaves does not help much.

Therefore, our algorithm is very beneficial until 16 slaves, but it is much less beneficial to go from 16 slaves to 32 or 64 slaves.

Table 3 gives the mean over 11 runs of the time taken to play the first move of a 9×9 game, for different numbers of total slaves, different numbers of slaves

Table 3. Time in seconds of the first move

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer							
100,000 playouts	65.02	32.96	16.78	8.23	4.49	—	—
50,000 playouts	32.45	17.05	8.56	4.08	2.19	—	—
2 slaves per computer							
100,000 playouts	—	35.29	17.83	9.17	4.61	3.77	—
50,000 playouts	—	16.45	9.23	4.61	2.25	1.74	—
4 slaves per computer							
100,000 playouts	—	—	20.48	13.13	5.47	3.77	3.61
50,000 playouts	—	—	10.33	6.13	2.82	1.83	1.75

Table 4. Time-ratio of the first move for 100,000 playouts

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer	1.00	1.97	3.87	7.90	14.48	—	—
2 slaves per computer	—	1.84	3.65	7.09	14.10	17.25	—
4 slaves per computer	—	—	3.17	4.95	11.89	17.25	18.01

Table 5. Time-ratio of the first move for 50,000 playouts

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves	64 slaves
1 slave per computer	1.00	1.90	3.79	7.95	14.82	—	—
2 slaves per computer	—	1.97	3.52	7.04	14.42	18.65	—
4 slaves per computer	—	—	3.14	5.29	11.51	17.73	18.54

per computer, and different numbers of playouts. The values were computed on an homogeneous network of dual cores. The associated variances are very low.

We define the speedup for n slaves as the division of the time for playing the first move with one slave by the time for playing the first move with n slaves.

Table 4 gives the speedup for the different configurations and 100,000 playouts, calculated using Table 3. Table 5 gives the corresponding speedups for 50,000 playouts. The speedups are almost linear until 8 slaves with one slave per computer. They start to decrease for 16 slaves (the speedup is then roughly 14), and stabilize near to 18 for more than 16 slaves.

A third conclusion we may draw from these tables is that it does not make a large difference running one slave per computer, two slaves per computer or four slaves per computer (even if processors are only dual cores).

In order to test if the decrease in speedup comes from the client or from the server, we made multiple tests. The first one consists in not playing playouts in the slaves, and sending a random value instead of the result of the playout. It reduces the time processing of each slave to almost zero, and only measures the communication time between the master and the slaves, as well as the master processing time.

Table 6. Time in seconds of the first move with random slaves

	1 slave	2 slaves	4 slaves	8 slaves	16 slaves	32 slaves
1 slave per computer						
100,000 playouts	25.00	12.50	6.25	4.44	4.10	—
2 slaves per computer						
100,000 playouts	—	12.49	6.94	4.47	4.48	3.93
4 slaves per computer						
100,000 playouts	—	—	6.26	4.72	4.07	3.93

Table 7. Time in seconds of the random master

100,000 playouts	2.60
50,000 playouts	1.30

The results are given in Table 6. We see that the time for random results converges to 3.9 seconds when running on 32 slaves, which is close to the time taken for the slaves playing real playouts with 32 or more slaves. Therefore the 3.9 seconds limit is due to the communications and to the master processing time and not to the time taken by the playouts.

In order to test the master processing time, we removed the communication. We removed the send command in the master, and replaced the reception command with a random value. In this experiment the master is similar to the previous experiment, except that it does not perform any communication. Results are given in Table 7. For 100,000 playouts the master processing time is 2.60 seconds, it accounts for 78% of the 3.3 seconds limit we have observed in the previous experiment.

Further speedups can be obtained by optimizing the master part, and from running the algorithm on a shared memory architecture to reduce significantly the communication time.

Table 8 gives the time of the parallel algorithm for various numbers of slaves, with random slaves and various fixed playout times. In this experiment, a slave sends back a random evaluation when the fixed playout time is elapsed. The first column of the table gives the fixed playout time in milliseconds. The next columns gives the mean time for the first move of a 9×9 game, the numbers in parentheses give the associated variance, each number corresponds to ten measures.

We see in Table 8 that for slow playout times (greater than two milliseconds) the speedup is linear even with 32 slaves. For faster playout times the speedup degrades as the playouts go faster. For one millisecond and half a millisecond, it is linear until 16 slaves. The speedup is linear until 8 slaves for playout time as low as 0.125 milliseconds. For faster playout times it is linear until 4 slaves.

Slow playouts policies can be interesting in other domains than Go, for example in General Game Playing. Concerning Go, we made experiments with a fast playout policy, and we succeeded parallelizing it playing multiple playouts

Table 8. Time of the algorithm with random slaves and various playout times

time	1 slave	4 slaves	8 slaves	16 slaves	32 slaves
10	1026.8 (6.606)	256.2 (0.103)	128.1 (0.016)	64.0 (0.030)	32.0 (0.020)
2	224.9 (0.027)	56.3 (0.162)	28.1 (0.011)	14.0 (0.005)	7.0 (0.002)
1	125.0 (0.081)	31.2 (0.006)	15.6 (0.001)	7.8 (0.006)	4.3 (0.035)
0.5	75.0 (0.026)	18.8 (0.087)	9.4 (0.001)	4.8 (0.034)	4.0 (0.055)
0.25	50.0 (0.005)	12.5 (0.024)	6.2 (0.001)	4.1 (0.019)	3.9 (0.049)
0.125	37.5 (0.021)	9.4 (0.001)	4.7 (0.007)	4.1 (0.222)	3.9 (0.055)
0.0625	25.0 (0.012)	6.6 (0.013)	4.4 (0.013)	4.0 (0.023)	3.8 (0.016)
0.03125	25.0 (0.007)	6.3 (0.004)	4.5 (0.110)	4.2 (0.0025)	4.0 (0.054)
0.01	25.0 (0.007)	6.3 (0.004)	4.5 (0.110)	4.5 (0.025)	4.0 (0.054)

Table 9. Results of the 8-slaves program against 16-slaves program

8-slaves with 50,000 playouts against 16-slaves with 100,000 playouts	33.50%
8-slaves with 25,000 playouts against 16-slaves with 50,000 playouts	27.00%
8-slaves with 12,500 playouts against 16-slaves with 25,000 playouts	21.50%

at each leaf. For 19×19 Go, playouts are slower than for 9×9 Go, therefore our algorithm should better apply to 19×19 Go.

The last experiment tests the benefits of going from 8 slaves to 16 slaves assuming linear speedups. Results are given in Table 9. There is a decrease in winning percentages as we increase the number of playouts.

5 A Summary of Findings

We have presented a parallel Monte-Carlo Tree Search algorithm. Experimental results against GNU Go 3.6 show that the improvement in level is efficient until 16 slaves. Using 16 slaves, our algorithm is 14 times faster than the sequential algorithm. On a cluster of computers the speedup varies from 4 to at least 32 depending on the playout speed. Using 5 seconds per move the parallel program improves from 40.5% with one slave to 70.5% with 16 slaves against GNU Go 3.6.

References

1. Brügmann, B.: Monte Carlo Go. Technical report, Physics Department, Syracuse University (1993)
2. Campbell, M., Hoane Jr., A.J., Hsu, F.-h.: Deep Blue. Artifial Intelligence 134(1-2), 57–83 (2002)
3. Cazenave, T., Jouandeau, N.: On the parallelization of UCT. In: Computer Games Workshop 2007, Amsterdam, The Netherlands, June 2007, pp. 93–101 (2007)
4. Coulom, R.: Efficient selectivity and back-up operators in Monte-Carlo tree search. In: CG 2006. LNCS, vol. 4630, pp. 72–83. Springer, Heidelberg (2006)

5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp. 97–104 (September 2004)
6. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML, pp. 273–280 (2007)
7. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
8. 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)

Using Artificial Boundaries in the Game of Go

Ling Zhao and Martin Müller

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

Abstract. Local search in the game of Go is easier if local areas have well-defined boundaries. An artificial boundary consists of temporarily added stones that close off an area. This paper describes a new general framework for finding boundaries in a way such that existing local search methods can be used. Furthermore, by using a revised local UCT search method, it is shown experimentally that this framework increases performance on local Go problems with open boundaries.

1 Introduction

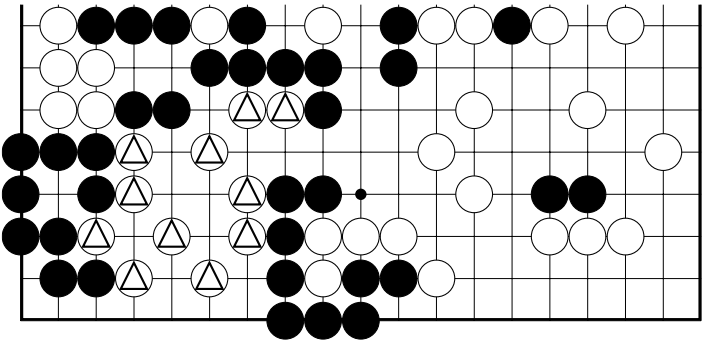
A local problem in Go assumes that (1) a goal defined on a closely related area is independent from the rest of the board, and (2) local search can be performed within this area in order to find the best moves for the goal. For example, such goals can be life and death of a group, connection of two blocks, or expansion of territory.

When local areas are not closed off, *soft* boundaries, which may contain empty points, have been used in many Go programs to define local Go problems. A soft boundary only defines the area relevant for the local problem. However, empty points on the boundary cause difficulties for local search. Some search extensions such as quiescence search may go beyond such a boundary on a contingency basis [1,2]. No general methods to define good soft boundaries for local problems have been found yet. A too tight boundary may make originally safe stones look dead due to reduced eye space, and a too loose one might make the local problem too big to solve in a reasonable time.

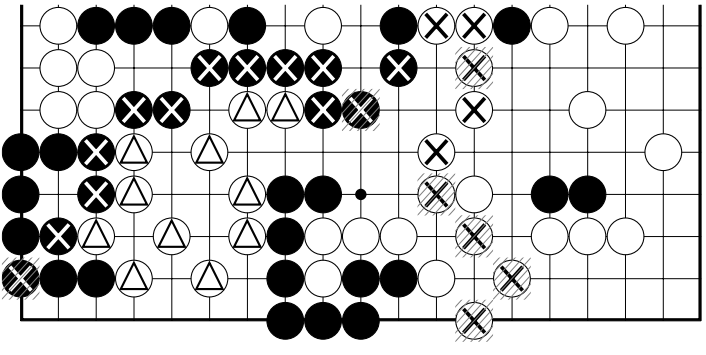
Some well-defined search problems such as capturing races [9], tsumego [7,12], and safety of territory [10] require a *hard* boundary consisting only of safe stones (there are certain extensions to allow external liberties). A hard boundary ensures that the local area is surrounded completely by safe stones. However, this happens rarely in real Go games, typically only in a very late phase of a game.

If a local area does not have a hard boundary, local search has to decide how to deal with points on the boundary of the area which have unknown safety status. By default, any blocks connected to an empty point or a stone of the same color on the boundary would be considered safe, since local search cannot go beyond the boundary, and any stones not killed at the end of local search are assumed safe. This naive approach does not work well in practice.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 81–91, 2008.



Original position. Local group stones are marked by △.



Modified position with artificial boundary added. The boundary is marked by ×. Shaded stones are added as an artificial boundary.

Fig. 1. Example of an artificial boundary

One solution to this problem is to temporarily add stones to the boundary in order to convert a soft boundary to a hard one by occupying all empty boundary points, or at least to reduce the number of empty points on a boundary as much as possible. Local search can be performed on this modified local problem with the assumption that all stones on the boundary are safe. After the search is finished, the added stones are removed to restore the original state. Figure 1 shows an example. To the authors' knowledge, this concept of artificial boundary is novel.

Artificial boundaries provide a general framework for utilizing existing local search methods that depend on a fixed, well-defined boundary. They can be used with traditional mini-max search, df-pn search [6], or UCT search [8]. In our experiments, we have successfully incorporated a local UCT search method into the artificial boundary framework to solve local group problems with open boundaries.

The remaining part of this paper is organized as follows. Section 2 describes the methods to find boundaries for a local area, and generate an artificial boundary if necessary. Section 3 introduces a revised local UCT search method designed to

work with artificial boundaries. Section 4 presents the experimental results, and Sect. 5 concludes the paper and discusses some possible future improvements.

2 Boundary and Artificial Boundary

In this section, we consider the following *boundary problem* in Go. Given a set of points and a goal related to it, find a boundary to surround the points such that the best moves for the goal can be found by a local search within the boundary. Stones added as an *artificial boundary* are assumed safe.

Figure 2 presents an algorithm *FindBoundary*() for finding a boundary for a set of points S. The method relies on an *oracle* such as an existing Go program that classifies groups and identifies their safety status.

```
1. // Input: S (point set)
2. // Output: OB, AB, EB (occupied, artificial, and empty boundaries)
3. // Preset parameters: InsidePointsLimit, RoundLimit
4. // BlockAt(p) returns the block which the occupied point p belongs to
5. FindBoundary(S, &OB, &AB, &EB)
6. {
7.     I = φ; round = 0; // I: inside points, round: expansion round
8.     while (Size(I) ≤ InsidePointsLimit)
9.     {
10.        S' = S; round++;
11.        Expand(S); // expand S outwards once in 4 directions
12.        S_e = S - S'; // expanded points
13.        if (IsEmpty(S_e)) break;
14.        foreach (p in S_e) // examine every point in S_e
15.        {
16.            if (Occupied(p) && InSafeGroup(p, &group))
17.                OB[group.Color()] += p; // to occupied boundary
18.            else if (NotOccupied(p) && InAtari(BlockAt(p)))
19.                I += BlockAt(p); // to inside points
20.            else if (NotOccupied(p) && InSafeGroup(p, &group)
21.                    && round ≥ RoundLimit
22.                    && NotAdjacentToStonesIn(I)
23.                    && NotAdjacentToOppStones(group.Color()))
24.                AB[group.Color()] += p; // to artificial boundary
25.            else I += p; // to inside points
26.        } // foreach
27.    } // while
28.    EB = FindEmptyBoundary(I); // to empty boundary
29.    VerifyBoundaries(&OB, &AB, &EB, &I);
30. }
```

Fig. 2. Procedure to create boundaries

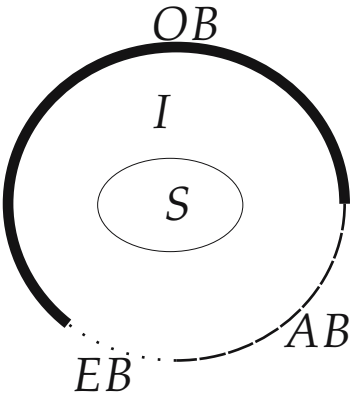


Fig. 3. Illustration of original area S, inside points I, and boundaries

The sets OB, AB, and EB contain the occupied, artificial, and empty points of the resulting boundary, respectively. The whole boundary B is represented by $OB \cup AB \cup EB$. The set of *inside points* I is situated in the space between the original area S and the boundary B. $S \cup I$ contains all the points surrounded by the boundary B. Their relations are illustrated in Fig. 3. The following subsections explain technical details of *FindBoundary*().

2.1 Weak Stones

The fate of a set of points S is related to the fate of neighboring weak stones W of either color. Although the life and death status of S might not depend on W, if S can live independently, the amount of territory to occupy certainly will. Any weak or unstable stones encountered during the expansion process are added to the inside points I. Blocks with a small number of liberties are included in I as well, because sente moves against them may be available and may affect the result of the original area.

2.2 Distance Conditions

Three distance conditions are enforced for an empty point p in an artificial boundary AB. First, p must be a minimum distance of *RoundLimit* (a parameter) away from S (see Line 21 of Fig. 2). This condition guarantees a reasonable space between the original area and its enclosing boundary. It is designed to prevent the boundary from reducing the eye space too much, and to avoid giving blocks in S an easy connection to friendly stones on the boundary. The method also provides a level of fault tolerance in the case of wrong status evaluation of p by the oracle. Misclassifications on a closer boundary are far more severe than on a boundary many intersections away. For the same reason, the second condition enforces that points in AB are never adjacent to stones in I (see Line 22), and

the third condition makes sure no point in AB is adjacent to enemy stones on the board (see Line 23).

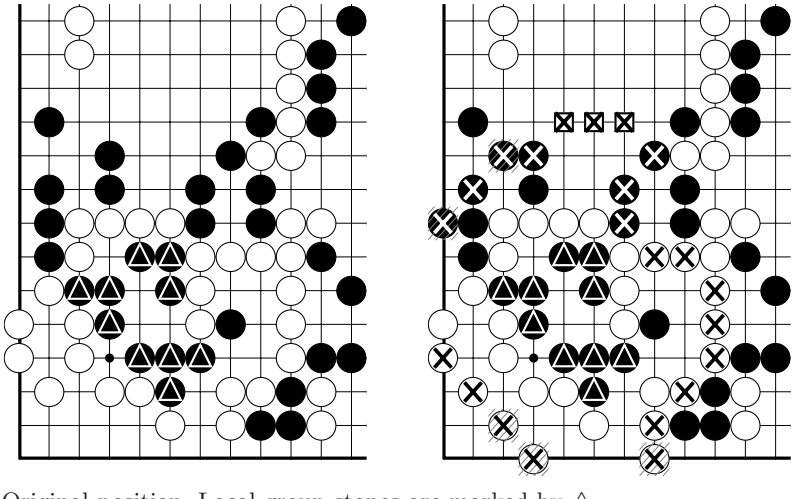
2.3 Expansion Stopping Conditions

In wide open problems, the expansion steps can easily include almost all points on the board, which contradicts the original purpose of localizing the problem. A hard limit ($InsidePointsLimit$) on the number of inside points is used to stop the expansion (see Line 8). In experiments, 40 seems to work well for local problems in 19×19 middle-game positions. If an expansion is stopped abruptly, the boundary might contain empty points, and they belong to the empty boundary EB. All points on EB are assumed to be safe, so any block reaching EB is treated as safe in the local problem. Figure 4 shows an example of an empty boundary.

An empty boundary models the escape option for a partially enclosed group. Once it jumps out of the gap represented by the empty boundary, it is assumed to be safe. If a local search method requires a hard boundary, EB can be filled by a series of stones of alternating color.

2.4 Boundary Verification

The last step in the algorithm is to verify the boundary. For example, a point that is originally classified as a boundary point might become an inside point later



Left: Original position. Local group stones are marked by △.
Right: Modified position with artificial and empty boundaries created. The boundary is marked by ×. Shaded stones are added as an artificial boundary. Empty boundary is denoted by ⊠.

Fig. 4. Example of an empty boundary

on. Similarly, a point considered as an inside point would become a boundary point if the expansion stops after the current round finishes. The condition at Line 23 ensures any artificially added stone has liberties, and thus adding stones of an artificial boundary is always legal.

3 Local UCT Search

UCT [8] is a Monte-Carlo sampling-based tree search algorithm that has become the backbone of the current best Go programs [4], and works especially well on small boards such as 9×9. UCT is usually used for *global* search. The local UCT search method is based on a standard MoGo-style UCT search [4]. The following two modifications are made.

1. Moves can only be generated within the area surrounded by the boundary. Thus random move generation must come from the area, and moves generated from heuristics such as atari moves or pattern moves need to be verified before being played out.
2. In global UCT, the evaluation of a terminal position is simply an overall win or loss. For local problems, the evaluation can be changed in two possible ways:
   (a) keep the win/loss evaluation, but evaluate whether a local goal is fulfilled or not;
   (b) use territory evaluation to measure the percentage of a player's points in the local area.

4 Experimental Results

All the experiments below were run on a Linux machine with a single core 3.4GHz Pentium CPU and 512M memory. Due to the stochastic nature of UCT, all results of local UCT search are the average of 10 runs.

4.1 Testing Local UCT Search – Tsumego

The default setting uses territory percentage to evaluate terminal positions in local UCT. Like the winning percentage in standard UCT, it is in the range of [0,1]. The performance of local UCT search is evaluated using a set of 50 tsumego problems created by Thomas Wolf's GoTools [11]. These problems are considered easy by brute-force solvers, taking less than one second per problem with a df-pn-based solver [7]. Local UCT search solves about 30 problems on average with 80,000 simulations, and it spent roughly 6 seconds per problem on average. Both win/loss and territorial evaluation yield similar results. The test set contains many ko fights, which is a disadvantage for purely local UCT search that lacks knowledge of external ko threats.

4.2 Testing Local UCT Search – Computer Go Game Positions

This test uses a collection of 50 test cases taken from Computer Go matches. They are all 19×19 middle-game full-board positions, and contain interesting local problems with open boundaries. The test set covers a range of local Go problems including life and death, connection, expansion, and seki problems. The test set can be found at [14].

The default settings in the experiments are as follows: $InsidePointsLimit = 40$, $RoundLimit = 4$, 10,000 simulations for local UCT with territorial evaluation being used. The Go program Explorer functions as the oracle to identify groups and estimate their status. Explorer is set to Level 1 which uses mainly static evaluation and no goal-directed search.

Average measurements over the 50 test cases are as follows: the size of S is 17.5, the size of I is 38.4, the whole area $S \cup I \cup B$ has a size of 77.4, and the number of empty points in the whole area is 38.8. The boundary length is 21.5, with 50.3% occupied, 32.2% artificial boundary, and 17.5% empty boundary. Artificial boundary helps to enclose 28 originally open local problems completely, and 22 problems still have empty boundaries present. There are 6 problems that have no artificial boundary at all.

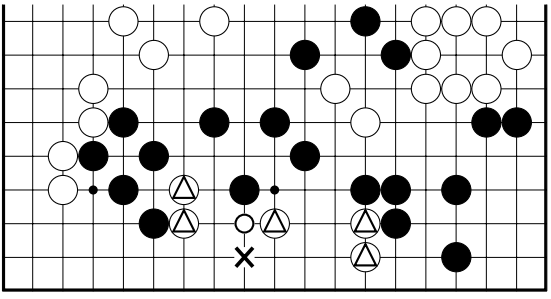
For comparison purposes, we use the Go programs Explorer and GNU Go 3.6 [5] in their default settings. They both have special commands to restrict generated moves to a given list of points. The whole area enclosed by the boundary computed by our program is used to form such a restricted point list.

AB+UCT is the program that uses both an artificial boundary and local UCT search. To measure AB+UCT against similar approaches, we use two extreme settings on local UCT for comparisons. NOAB+UCT measures how the program performs with artificial boundary disabled, but moves are still restricted within a certain area. It can be viewed as using the *FindBoundary*() procedure in Fig. 2 with *RoundLimit* set to infinity, so that no artificial boundary is generated when a boundary is created to surround a local area. NOAB+GUCT is the setting where UCT runs simulations on the whole board if no hard boundary can be found, but the evaluation of terminal positions is based on the territory score of the original area, not the full board. It is done by setting both *RoundLimit* and *InsidePointsLimit* to infinity in *FindBoundary*().

Table 1 below shows the results of solving local problems from 5 different approaches. Default settings were used, and the results were averaged over 10 runs. The AB+UCT approach had a comparable performance with GNU Go,

Table 1. Results of solving 50 local problems

Program	#Solved	Solved %	time (sec)
AB+UCT	40.4	80.8%	87.6
NOAB+UCT	36.9	73.8%	95.3
NOAB+GUCT	35.3	70.6%	462.6
Explorer	28	56.0%	453.4
GNU Go	43	86.0%	183.8



Local group stones are marked with △ with black to play. The best move and the move preferred by local UCT are marked with ○ and × respectively.

Fig. 5. Example of a failed case

but it spent significantly less time. The table also demonstrates that the artificial boundary has a positive impact on localizing the problem, which in turns helps to make the problem solvable by local UCT.

In the experiments, 16 problems were always solved, but there were no problems unsolved by all approaches. For AB+UCT, 4 problems failed consistently, and there were a total of 19 problems that failed in the 10 runs. For NOAB+UCT however, 7 problems always failed (including the 4 problems that always failed in AB+UCT), and 21 problems failed in some run. Of the 4 always failed cases in the AB+UCT experiment, one could be solved when the number of simulations is over 40,000, and two problems could be solved when the parameter *InsidePointsLimit* is increased to 50 and 70 respectively. For the remaining problem, as shown in Fig. 5, local UCT could never find the optimal move within 80,000 simulations.

The number of simulations has great impact on the strength of UCT. Figure 6 shows a parameter study varying the number of simulations. Since NOAB+GUCT takes too much time, it is not a practical method and was not included for this experiment. According to the first graph in Fig. 6, the number of simulations is closely tied to the strength of the program. An increase of simulations almost always results in an increase of the performance. In the case of AB+UCT, the increase became much less when the number of simulations reached 10,000. The performance also stabilized after that data point, where the standard deviations of the number of solved cases were below 1.2. The second graph in Fig. 6 clearly shows that AB+UCT spent negligible time creating the boundary, and the time it used was almost linear with the number of simulations in local UCT.

Experiments were run to determine the parameters *InsidePointsLimit* and *RoundLimit* in Fig. 2, and the results are summarized in Table 2. An increase of *InsidePointsLimit* or *RoundLimit* results in the increase of the area the boundary encloses, and as a result, local UCT spends more time if the number of simulations is fixed. For parameter *InsidePointsLimit*, values greater than

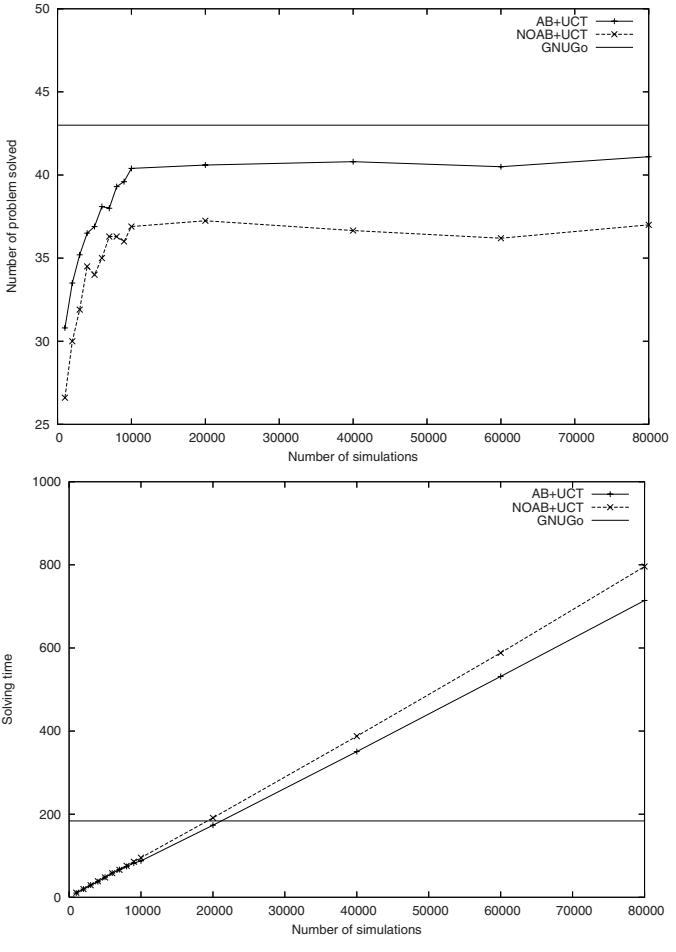


Fig. 6. Results of local problems w.r.t. the number of simulations

Table 2. Experiments on *InsidePointsLimit* and *RoundLimit*

Limit	10	20	30	40	50	60	70	80	90	100
#Solved	26.2	30.4	34.6	40.4	40.0	40.0	40.8	39.4	39.4	40.4
Time (sec)	42.4	58.08	71.7	87.6	97.38	102.76	111.56	115.32	121.04	126.0

Solving results w.r.t. *InsidePointsLimit*

Limit	1	2	3	4	5	6
#Solved	35.8	36.0	40.6	40.4	36.2	38.0
Time (sec)	68.32	73.26	79.6	87.6	89.9	91.7

Solving results w.r.t. *RoundLimit*

or equal to 40 yielded a similar performance in terms of the number of problems solved. For parameter *RoundLimit*, 3 and 4 achieved the best results.

One interesting discovery is about the area to be scored at terminal nodes of UCT. There are two choices for defining such an area: the original area S that denotes the local problem, or the whole local area $S \cup I \cup B$. Our experimental results on the test set strongly favor the first choice. Using the whole area to score resulted in a big degradation of performance on the test set. Note that this option only affects the evaluated area, and moves are generated for the whole area in both cases. We believe the main reason that evaluating S only works better on the test set is because the test set is made of partially enclosed local problems from middle-game positions.

Rapid Action Value Estimation (RAVE) [3] is an enhancement of UCT similar to the history heuristic that works very well in standard UCT. However, it did not work with local UCT. The number of unsolved cases increased from 9.6 to about 15 with 10,000 simulations. This requires further study.

5 Conclusions and Future Work

This paper investigated hard and soft boundaries for local Go problems, and introduced the concepts of occupied boundary, artificial boundary, and empty boundary. We then proposed a new general framework to create artificial boundaries for local Go problems that are not enclosed completely. From the experimental findings we provisionally may conclude that the artificial boundary framework combined with a local UCT search contributes to the progress of search techniques in computer Go in the future.

Artificial boundaries help to find a hard boundary to enclose the local problem completely, or decrease the number of empty points on the boundary to reduce uncertainty, so that local search methods can be used more effectively. In the experiments, Explorer was used for group identification and safety evaluation, and a local UCT search method was employed to find the best moves. Although artificial boundaries did have positive impact on the performance in the experiments, the AB+UCT approach still had many failed cases, since both Explorer and local UCT are far from perfect. It would be interesting to see if a domain-independent method can take Explorer's role in this approach, so that the artificial boundary framework can be applied in a broader domain. In addition, we would like to use the framework in conjunction with other local search methods, such as life-and-death and safety solvers, to conquer problems with open boundaries.

For future work, we would like to apply this framework to full board positions using a divide and conquer approach. Then it can utilize the probabilistic combinatorial game model [13] to focus on maximizing the overall winning probability when combining local results.

Acknowledgments. The authors would like to thank Markus Enzenberger for his implementation of the global UCT search and helpful discussions, and the

anonymous referees for many constructive comments. This research was financially supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, the Alberta Ingenuity Fund and iCORE, the Alberta Informatics Circle of Research Excellence.

References

1. Cazenave, T.: A generalized threats search algorithm. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 75–87. Springer, Heidelberg (2002)
2. Fotland, D.: Learning: Chess programs versus Go programs. Computer Go Mailing List (2004)
3. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: International Conference on Machine Learning (ICML 2007) (2007)
4. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modifications of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
5. GNU Go webpage, http://www.gnu.org/software/gnugo/
6. Kishimoto, A., Müller, M.: Df-pn in Go: An application to the one-eye problem. In: Advances in Computer Games, vol. 10, pp. 125–141. Kluwer Academic Publishers, Dordrecht (2003)
7. Kishimoto, A., Müller, M.: Search versus knowledge for solving life and death problems in Go. In: Twentieth National Conference on Artificial Intelligence (AAAI 2005), pp. 1374–1379 (2005)
8. 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)
9. Müller, M.: Race to capture: Analyzing semeai in Go. In: Game Programming Workshop in Japan 1999, vol. 99(14). IPSJ Symposium Series, pp. 61–68, Tokyo, Japan (1999)
10. Niu, X., Müller, M.: An improved safety solver for computer Go. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 97–112. Springer, Heidelberg (2006)
11. Wolf, T.: Gotools webpage, http://lie.math.brocku.ca/GoTools/
12. Wolf, T.: Forward pruning and other heuristic search techniques in tsume Go. Special issue of Information Sciences 122(1), 59–76 (2000)
13. Zhao, L., Müller, M.: Solving probabilistic combinatorial 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. 225–238. Springer, Heidelberg (2006)
14. Zhao, L., Müller, M.: Artificial boundary test set for Go (2008), http://www.cs.ualberta.ca/~games/go/abtest.zip

A Fast Indexing Method for Monte-Carlo Go

Keh-Hsun Chen, Dawei Du, and Peigang Zhang

Department of Computer Science, University of North Carolina at Charlotte, Charlotte, NC 28223, USA
{chen,ddu,pzhang1}@uncc.edu

Abstract. 3×3 patterns are widely used in Monte-Carlo (MC) Go programs to improve the performance. In this paper, we propose a direct indexing approach to build and use a complete 3×3 pattern library. The contents of the immediate 8 neighboring positions of a board point are coded into a 16-bit string, called surrounding index. The surrounding indices of all board points can be updated incrementally in an efficient way. We propose an effective method to learn the pattern weights from forty thousand professional games. The method converges faster and performs equally well or better than the method of computing "Elo ratings" [4]. The knowledge contained in the pattern library can be efficiently applied to the MC simulations and to the growth of MC search tree. Testing results showed that our method increased the winning rates of Go Intellect against GNU Go on 9×9 games by over 7% taking the tax on the program speed into consideration.

1 Introduction

The positional evaluation difficulty and the large branching factor have made Go the most challenging board game for AI research. MC tree search with the UCT algorithm [5,7] is the most effective approach known today in playing Go by a computer. Pattern knowledge is proven to be very helpful in both classical Go programs and MC Go programs. Almost all Go programs deal with patterns in some form. Several efforts on automatic local Go pattern generation and learning research have been made [1,2,8,9]. They typically concentrated on dealing with good/urgent patterns and ignored the vast majority other patterns. Most of those pattern libraries are intended to be used directly by the move decision of the actual program. However, they cannot provide sufficiently fast access for the MC simulation routines. To improve the quality of the simulation in MC Tree Search, Coulom [4] used an "Elo rating" computation approach to obtain the relative weights of different features of the candidate moves. Moreover, Gelly and Silver significantly improved the performance of the UCT algorithm [5,7] by combining online and offline knowledge [6]. In this paper, we perform offline pattern mining from forty thousand professional Go game records to create libraries of weights/urgencies of all local patterns in Go under a given restricted template, with 3×3 being the initial template size. Since we intend to use these weights in Monte-Carlo simulations, the pattern matching speed needs to be extremely

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 92–101, 2008.

fast. We design a direct indexing approach to access the weight information. The surrounding index of each board point is updated incrementally as moves are executed. We start by computing the adoption ratios of surrounding indices through processing forty thousand professional 19×19 Go game records. These pattern adoption ratios, taking rotations and symmetries into consideration, serve as the initial pattern urgency estimates. The pattern urgencies, or weights, are computed through additional iterations of non-uniform adoption rewards based on the urgency estimation values in the previous iteration of the competing points until the sequence of values converge. These weights are used in the MC simulations and in guiding the growth of the MC search tree. Testing results showed that this approach increased the winning rates of Go Intellect against GNU Go in 9×9 games by about 7.5% after the adjustment on the tax to the program speed due to the extra processing on the surrounding index information.

In Sect. 2, we introduce our surrounding index scheme. Sections 3 and 4 discuss details of our pattern mining. Section 5 shows the top patterns learned. Our method is compared with Coulom's computing "Elo ratings" method [4] in Sect. 6. We show how the new pattern knowledge is used in simulations in Sect. 7 and in guiding MC tree search in Sect. 8. Section 9 presents experimental results demonstrating the merits of the surrounding indexed weights. In Sect. 10, we discuss surrounding indices for some larger pattern templates. Section 11 concludes the paper with work in progress and future work.

2 Surrounding Index

Our goal is to create libraries of weights/urgencies of all local patterns in Go under a given restricted template with fast direct indexing access, so we can use the information to improve the play-outs and to help guiding the growth of the MC search tree. We implemented it first for a 3×3 pattern template. We use two bits to code the contents of a board point: empty (00), black (01), white (10), or border (11). The immediate surrounding pattern of a board point is a sequence of 8 2-bit codes from the north neighbor, the northeast diagonal, the east neighbor, etc., to the northwest diagonal, which can be coded as a 16-bit binary string called surrounding index (SI). After initialization, the surrounding indices of all board points can be updated incrementally as moves are executed. When a stone is added, it only affects the surrounding indices of its 8 immediate neighbors. For each of the 8 neighbors, the updating involves changing certain two 0-bits (empty) to the code for the color of the new stone. Similarly, when a stone is removed, certain 2 bits, coding the old color, will be reset to 00. We are only interested in the surrounding indices of empty board points, but we have to keep track of the surrounding indices of all board points since stones could be captured and their points become available empty points thereafter. The above updating can be implemented efficiently in the program's execute move and undo move routines with just a small overhead, 8% in our case, on the processing time. The above indexing framework can be extended to any

local pattern template. For example, we have also created 24-bit surrounding indices for neighbors within Manhattan distance 2. The time cost for updating surrounding indices increases only linearly to the template size. The resource bottleneck is the exponential memory space requirement, because we need to keep track of arrays of size 22s where s is the number of neighboring points in the template. In the next section, we discuss calculating adoption ratios for surrounding indices.

3 Adoption Ratios

We start our pattern-mining procedure by obtaining the adoption ratios for each possible surrounding index (local pattern) from professional Go game records. We use two arrays NuOccurrences[ ] and NuAdoptions[ ], both indexed by a surrounding index and initialized to 0. We put all game records to be processed in one directory, then load and process them one at a time. For each game record, we step through the moves played one move at a time with surrounding indices automatically updated. We compute the initial adoption ratios from black's point of view. So if black is to play, we increment NuOccurrences[i] by 1 for all surrounding indices i of legal points of the current board configuration and increment NuAdoptions[j] by one for the surrounding index j of the move chosen in the record. If White is to play, all the indices go through a procedure flipBW, which produces the corresponding surrounding index with black and white reversed.

We normally give capturing related moves higher weights than surrounding indexed weights in the overall move ranking. We do not count into the adoption ratio statistics if the chosen move is a capturing move, atari move, or an extension or connection move after a block being ataried, since those moves were played due to the urgencies of capture/escape, not because of the urgencies of the local patterns. We do not want our pattern weights to be distorted by capturing related moves, the small 3×3 template cannot tell whether it is a capture/escape case or not. Our MC simulation procedure treats capture/escape separately with higher priority before performing selection by surrounding indexed weights.

Each pattern has 8 equivalent patterns under rotations and symmetries (not counting the color flipping). We use a loop to add counts from equivalent patterns together to be shared by all patterns (surrounding indices) in the same equivalent class. The adoption ratio of a pattern is calculated as the number of adoptions divided by the number of occurrences of the corresponding surrounding index:

$$AdoptionRatio[i] = NuAdoptions[i]/NuOccurrences[i] \text{ for each SI } i \ . \quad (1)$$

The adoption ratios form our initial weights.

4 Pattern Weights

A pattern (surrounding index) with high adoption ratio is not necessarily an urgent pattern. It may occur when the competing patterns on the board are

all weak, then a number of adoptions is registered, but they may not be really urgent. In contrast when the board has several urgent patterns occurring, we wish to award the adopted pattern a higher credit for "beating" those tough competitions. So, we do a second pass, this time for each pattern selected to play according to the game record. Its weight is increased by the adoption ratios of those other patterns that occurred in the board configuration, which means when the move with SI i is selected to play:

$$W_{new}(i) = \sum_{j \neq i \text{ and } j \text{ occurs on the board}} W_{old}(j) \quad (2)$$

At the end of a pass, we share weights within an equivalent class of patterns. Each member obtains the sum of the pattern weights within the class. After each weight is divided by the pattern equivalent class occurrence count, we normalize the new weights by making their total 64K. We repeat this process many times until the weights converge, i.e., they do not change much from one iteration to the next. We view a weight distribution over all surrounding indices as a point in the 2^{2s} dimensional space. When the distance $d = \sqrt{\sum_i (w_i - w_i')^2}$ between two consecutive passes is less than a threshold (which we set to 1), where w_i and w_i' are the weights of surrounding index i at two consecutive passes, the weights are considered stabilized. We apply this method to the collection of forty thousand professional games. After initializing the weights to adoption ratios, it took just 4 iterations for the weights to converge. Each iteration took about 20 minutes. Experiments showed the converged weights perform better than the original adoption ratios.

5 Top Patterns

Our pattern mining calculates the weights of all 64K surrounding indices. Among them, only 7857 indices correspond to legal 3×3 patterns with an empty point in the center. We have 3^8 indices with the center point away from the edges of the board, $3^5 * 4$ indices with the center on line 1 & not a corner, and $3^3 * 4$ indices for the corners. 7208 of the 7857 legal surrounding indices actually occurred in the forty thousand professional games. The 649 legal surrounding indices that never occurred in the 40 thousand professional games were given a very low default weight.

Figure 1 shows the top 10 pattern groups of highest weights. Only one representative is listed from any equivalent pattern group. The top row shows the 10 patterns with highest adoption ratios. The leftmost one on the top row has a slightly higher than one forth adoption ratio in the professional game set. After four iterations, the final top ten patterns are on the bottom row. The highest weight pattern, the leftmost pattern in Fig. 1, has a weight 122.5; the 10^{th} highest weight pattern, the rightmost pattern in Fig. 1 has a weight 72.39.

In the next section, we shall consider an alternative way to learn the pattern weights.

6 "Elo Ratings"

In our model for computing "Elo ratings", the surrounding indices that occurred on a board configuration are considered in a multi-way competition and the SI of the move played in the game by the professional player is the winner of the competition. We do not have the complex situation of teams of features as in Coulom's paper [4] to deal with. The minorization-maximization formula in [4] can now be simplified to the following weight updating formula.

$$W_{new}(i) = \frac{NuAdoptions(i)}{\sum_{\text{i occurs on the board}} \frac{1}{\sum_{\text{p is a legal point on the board}} W_{old(SI(p))}}} \quad (3)$$

Originally we assume all SI having a weight 1. Then the first-round calculation computes

$$W_{new}(i) = \frac{NuAdoptions(i)}{\sum_{\text{i occurs on the board}} 1/\text{Number legal moves on the board}}$$

The top 10 of this set of weights are shown on the top row of Fig. 2. It took 9 additional iterations to converge (weight vector distance to the weight vector from previous run less than 1). The time needed for each iteration is about the same as our method. The final top-10 patterns are shown in the bottom row of Fig. 2.

Comparing Figs. 1 and 2, we can see that the two sets of leading patterns are surprisingly similar. The two sets of the initial top 10s have 9 patterns in common and the two sets of the final top 10s are the same, just with orders slightly rearranged.

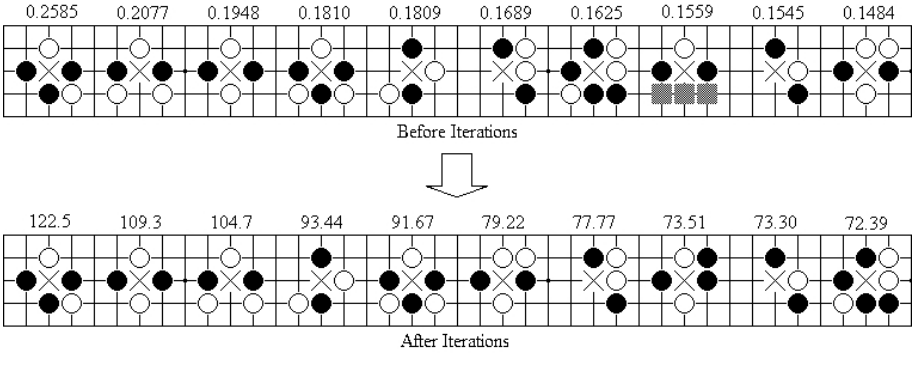


Fig. 1. The 10 patterns with highest weights/urgencies based on the 8 immediate surroundings. We list one representative from an equivalent class. It took only 4 iterations to converge. All patterns are 3×3 with Black to play in the center X. The weights are marked above the patterns. The dark shades mark the border, which means that the two black stones just above the dark shades are on line 1.

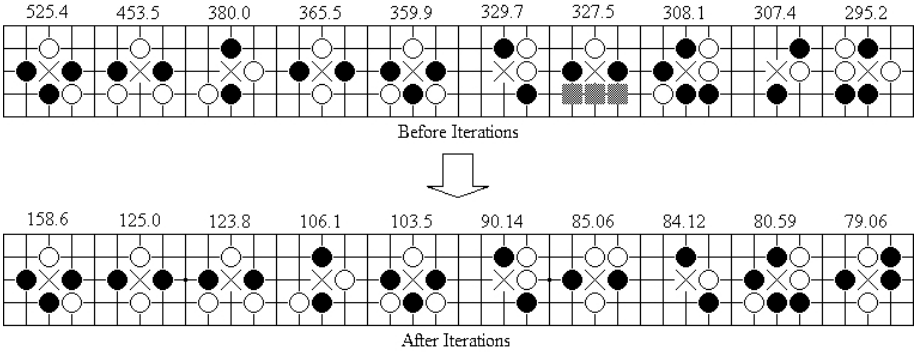


Fig. 2. Calculating "Elo rating" takes 9 iterations to converge

Before we compare their performances in Sect. 9, we shall discuss how we use the surrounding indexed weights in Go Intellect. We discuss their use in simulation play-outs in Sect. 7 and in guiding the MC Tree Search in Sect 8.

7 Monte-Carlo Simulation

We explored many different ways to use these local surrounding weights in the simulations for Monte-Carlo Tree Search. We found that the following priority order on move generators performed the best.

1. Handling urgent capture and escape.
2. Play urgent pattern move near the last opponent move.
3. Play based on the weights indexed by the surrounding indices.
4. Random sweeping with neighborhood replacement.

Urgent capture and escape is our top priority item in the simulation. The capture/escape situation is checked only for the last move itself and its adjacent opponent blocks. Pseudo ladders are performed to determine the urgencies. The second priority is given to the urgent pattern moves near the last move. The third priority relies on our fast indexing method (see below). The random sweeping is the last resort in generating a move. It randomly picks a board point as the starting point, and then sequentially scans every empty point encountered and plays at the first legal point encountered. The neighborhood replacement allows it to replace an obvious bad move by a more reasonable one, such a move on an empty neighbor or diagonally legal point. For details of these move generators, we refer to [3]. Now, we shall discuss the move generator based on the surrounding indexed weights. First a linear search is performed to find the highest weight for legal points on the board. If the highest weight is less than the cutoff weight of (4) then go to generator 4. Otherwise we use the 90% of the highest weight as a threshold. Subsequently, we select up to 10 candidate moves. If the highest weight is less than 10, up to 10 empty moves are added as candidates. Then we randomly

pick a move from the set of candidates. If an empty move is picked then generator 4 will generate a move. The cutoff weight, the threshold, and the limited number of candidate moves were determined after tedious experiments and tuning over time. It is very important to append empty choices to increase the randomness. Without empty choices, the program performance drops significantly.

8 Monte-Carlo Tree Search

We use the surrounding indexed weights in guiding the MC Tree Search in two ways: (1) as contributing to move ordering and (2) as a prior knowledge [6].

When we expand a node in the MC search tree it is desirable to add child nodes in priority order. Capturing and local urgent patterns can select the top few children. After that, the surrounding indexed weights play a deciding role in ordering the rest moves.

We also use surrounding indexed weights along with weights from capturing/escape and urgent patterns around last move to form prior knowledge about a move [6]. We modified the UCT algorithm as follows. In stead of selecting the child node i maximizing $w_i/n_i + \sqrt{\lg(p)/(C * ni)}$ where w_i is the number of winning simulations passing through child node i, n_i is the total number of simulations passing through child node i, p is the total number of simulations passing through the parent node, and C is a constant (we use 5), we select the child node i maximizing $(w_i + n_q)/(n_i + n_p) + \sqrt{\lg(p)/(C * ni)}$.

Intuitively, based on prior knowledge we treat the child node i as if there were n_p extra games already played and winning n_q of them. After some experiments, we selected 30 to be n_p. Let n_q be dynamically determined by (1) the grandparent winning rates, (2) the parent winning rates, and (3) the relative weight of the move to the max weight among all the siblings.

9 Experimental Results

We ran three performance comparison tests against GNU Go 3.6 on three versions of Go Intellect based on the UCT algorithm with MC Tree Search.

GI0 - using move generators 1, 2, & 4
GIS - using move generators 1, 2, 3, & 4 with the converged weights
GIE - using move generators 1, 2, 3, & 4 with the "Elo ratings"

Each version played at least 1200 9×9 games against GNU Go 3.6 level 10 with various upper limits on the number of simulations per move. Adding move generator 3 slowed down the simulation play-outs by about 28% including the cost of updating surrounding indices and extra processing cost of move generator 3. We test GI0 with the following number of simulation limits: 12.5K, 50K, 100K, 200K, and 400K. To make it a fair test, we tightened the limits for GIS and GIE by 28% to the actual settings of 9K, 18K, 36K, 72K, 144K, and 288K, respectively. Each program played Black half of the times and White the other

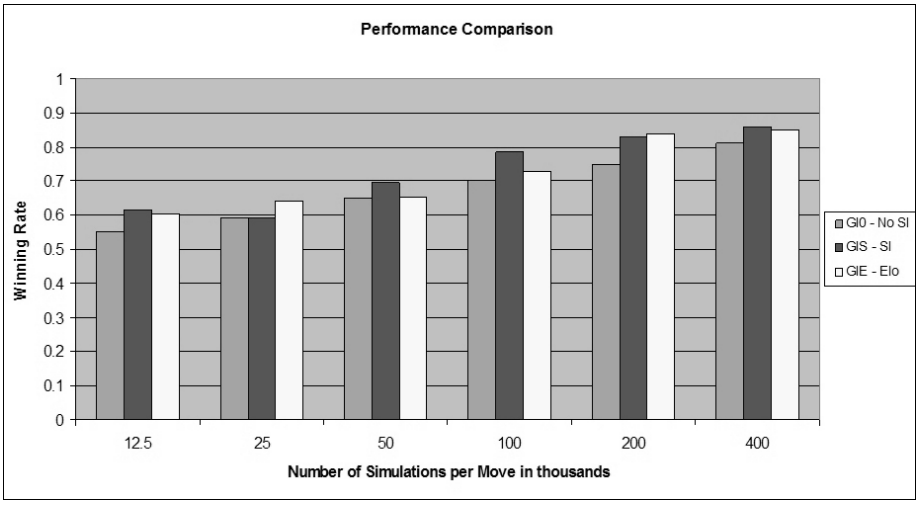


Fig. 3. Comparison of the performances of three versions of Go Intellect: (1) GI0 does not use SI, (2) GIS uses SI with weights learned by our algorithm, and (3) GIE uses "Elo ratings". We reduced the limit of the number of simulations per move by 28% in GI1 and GI2 to make it a fair comparison since GIS and GIE need 28% overhead in keeping track of SI and using its weights. For example, in the first set GIS and GIE had the limit of the number of simulations per move set at 9000 to compare GI0 with 12,500 simulations per move.

half of the times. We ran our experiments mostly on a 136-node cluster; each node is a PowerPC G5 at 2.33GHz. Figure 3 compares the performances of the 3 versions of Go Intellect against GNU Go on 9×9 games.

The contribution of the surrounding index scheme is significant. It improves the winning rates against GNU Go in 9×9 games by over 7% on average with the converged weights. Our weights slightly outperform "Elo ratings", but the difference is within the margin of statistical error. Therefore, we will say they perform equally well.

We also tried to use the weights as a probability distribution to pick a move (semi) randomly. This increased the winning rates on a given limit on the number of simulations per move, but it further slowed down the simulation speed by another 50% and it took twice as long to make a move decision.

10 Extended Surrounding Indices

We explored several extensions to the 16-bit surrounding index. One natural extension is to use the template of all points within a Manhattan distance of 2 (MD2), which add 4 extra points to the pattern template and extend the surrounding index to 24 bits. We use similar methods to update incrementally the 24-bit surrounding index of all board points and to compute the indexed

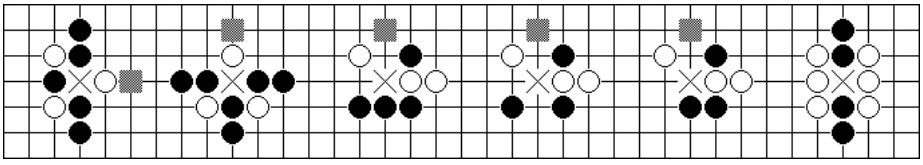


Fig. 4. Top six MD2 patterns. Dark shade marks border.

weights. Figure 4 shows the top 6 MD2 patterns after the weight convergence. The result was not nearly as good as the simple 16 bit surrounding index. We then modified the scheme to use the extra 8 bits with 2 bits in a group to code the number of liberties of an immediate neighbor (00: 1 liberty, 01: 2 liberties, 10: 3 liberties, 11: 4 or more liberties or two solid eyes). If the immediate neighbor is empty then we use the two bits to code the contents of the board point further away at MD2. We still could not achieve good results. The reality is that changing from 16-bit indices to 24-bit indices, the number of indices under consideration increased 256 fold but we did not have 256 times as many professional games to process with and to learn from - we only had the same 40 thousand game records. Each surrounding index has much lower frequency of occurrences, which makes weight leaning difficult.

We also tried a compromise: a 20-bit surrounding index scheme. Using each of the extra 4 bits to code the number of the liberties of an immediate neighbor (1: 4 or more liberties or 2 or more solid eyes, 0 otherwise). If the immediate neighbor is empty, it indicates whether the further away neighbor is empty or not. Here, we introduced some weight inference rules. For example, with the same pattern stone layout, lower number of liberties implies higher urgency. As a heuristic rule, the higher urgency SI gets to increase its weight by the weight of the lower urgency SI with the same stone layout. The situation here is not nearly as sparse as 24 bits. But since it needs to access continuously the liberty count information, it further slows down the simulation by an additional 25 to 30%. The experimental results still do not compare favorably to those from the simple 16-bit surrounding index scheme.

11 Conclusion and Future Work

Two lessons learned from the surrounding index project are that in a Monte-Carlo simulation, (1) simplicity has its merit and (2) randomness is important. Accessing local Go patterns, learned from professional Go games, through fast accessing and updating surrounding indexing is an effective way in improving the strength of MC-Go programs. Testing on a 13×13 and a 19×19 Go is the next item on our agenda.

Using the surrounding indexed weights in progressive widening [4] will be investigated in the near future. Since the surrounding indexed weights provide reasonable urgency estimates on all legal moves. We could take advantage of it in the progressive widening scheme that as the number of simulation games passing

through the parent node grows, we increase total child weight limit instead of total child count limit.

When combined with additional parameters, such as the distance to the board edge, the stage of the game, the number of liberties of a candidate move, and the distance to the last move, we can hopefully further unleash the potential power of the surrounding index scheme. We are accumulating computer testing games, so in the future we can have a sufficiently large volume of game records to support a larger indexing scheme.

References

1. Bouzy, B., Chaslot, G.M.J.-B.: Bayesian Generation and Integration of K-nearest neighbor Patterns for 19 × 19 Go. In: Kendall, G., Lucas, S. (eds.) IEEE 2005 Symposium on Computational Intelligence in Games, Essex, UK, pp. 176–181 (2005)
2. Bouzy, B., Chaslot, G.M.J.-B.: Monte-Carlo Go Reinforcement Learning Experiments. In: IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA, pp. 187–194 (2006)
3. Chen, K., Zhang, P.: Monte-Carlo Go with Knowledge-guided Simulations. ICGA Journal 31(2), 67–76 (2008)
4. Coulom, R.: Computing "Elo Ratings" of Move Patterns in the Game of Go. ICGA Journal 30(4), 198–208 (2007)
5. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA (2006)
6. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference of Machine Learning (ICML 2007), pp. 273–280 (2007)
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. Silver, D., Sutton, R.S., Müller, M.: Reinforcement Learning of Local Shape in the Game of Go. In: 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 1053–1058 (2007)
9. Stern, D., Herbrich, R., Graepel, T.: Bayesian Pattern Ranking for Move Prediction in the Game of Go. In: Proceedings of the 23rd International Conference on Machine Learning, Pittsburgh, PA (2006)

An Improved Safety Solver in Go Using Partial Regions

Xiaozhen Niu and Martin Müller

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

Abstract. Previous safety-of-territory solvers for the game of Go have worked on whole regions surrounded by stones of one color. Their applicability is limited to small to medium-size regions. We describe a new technique that is able to prove that parts of large regions are safe. By using pairs of dividing points, even huge regions can be divided into smaller partial regions that can be proven much easier and faster. Our experimental results show that the new technique significantly improves the performance of our previous state of the art safety-of-territory solver. Especially in earlier game phases, the solver utilizing the new technique outperforms the previous solver by a large margin.

1 Introduction

Evaluating the safety of territories is one of the most important components of a Go program. The previous work in [4,5,6] introduces several search-based safety-of-territory solvers that can determine the correct safety status of a given region. One weakness of the previous solvers is that their applicability is limited to small to medium-size regions. The reason is that the search space grows exponentially with region size. In real games, most often a board contains very large regions. When more and more stones are played, these large regions are gradually divided into smaller regions. Therefore the safety-of-territory solver can only be applied in the late stage of a game.

This paper introduces a new technique that can prove the safety of parts of large regions. By applying a miai strategy to pairs of dividing points, a large region can be divided conceptually into smaller partial regions. Separate safety searches can then be performed on each of these smaller partial regions. The experimental results show that the partial proving module improves the performance of a state of the art safety-of-territory solver. Even early in games, when there are only large regions on the board, the current system can prove the safety of many partial regions and their surrounding blocks.

The structure of this paper is as follows. Section 2 briefly discusses related work. Section 3 explains details of the partial proving module. Section 4 discusses experimental results, and the final section provides conclusions and further research directions.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 102–112, 2008.

2 Related Work

There are many successful approaches for safety recognition proposed in the literature. The classical algorithm due to Benson statically recognizes *unconditionally alive* blocks and regions on board [1]. A number of papers address the question of eye shape of a region surrounded by a single block. Vilà and Cazenave's static classification rules evaluate many such regions of size up to 7 points as safe [7]. Dyer's eye shape library contains eye shapes up to size 7 [2]. Wolf and Pratola extend the analysis to size 11 regions, and compute many interesting properties of such regions such as ko status [9].

Müller identifies regions that are safe by alternating play [3]. The work introduces the notion of miaipairs for statically proving the safety of regions that can make two eyes in two independent ways. Van der Werf presents a learning system for high-accuracy heuristic scoring of final positions in the game of Go [8].

The current work extends the safety solvers described in [4,5]. *SAFETY SOLVER 1.0* is the previously best solver for evaluating the safety of completely enclosed regions. It can solve regions with size up to 18 empty points in reasonable time. *SAFETY SOLVER 2.0*, described in [5], can handle open boundary regions. Its board partitioning is based on open boundary zones. Its size limitation is similar. The current paper focuses on recognizing safe partial regions inside large regions with no size limitation.

3 Using Partial Regions for Safety Recognition

This section describes the four major processing steps and the related algorithms that are implemented to prove partial regions safe.

3.1 Find Dividing Miaipairs

To prove parts of a region R as safe, it must first be divided into reasonable chunks. A simple miai strategy is utilized for this purpose. A *miaipair* [3] is a pair of two empty points inside R, such that the defender playing at either of these two points would split R into two subregions. A defender miai strategy applied to these two points forces the defender to occupy at least one of these points: whenever the attacker plays one point and the other one is still empty, the attacker is forced to reply there.

This paper focuses on miaipairs containing two adjacent points which are also adjacent to defender boundary stones. Let $L(R)$ be the set of all splitting points inside R which are liberties of boundary blocks of the region. In the example on the left of Fig. 1, $L(R) = \{c1, e1, f1, j1\}$. The only miaipair is $\{e1, f1\}$. The black region on the right of Fig. 1 contains two overlapping miaipairs $\{d1, e1\}$ and $\{e1, f1\}$.

3.2 Dividing a Single Region Using One Miaipair

The simplest approach uses a single miaipair to divide a region R. Figure 2 shows a large black region R with miaipair $P = \{o1, p1\}$. By following the miai

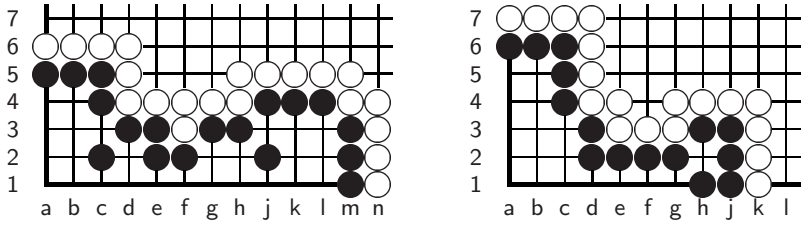


Fig. 1. Examples of miaipairs inside black regions

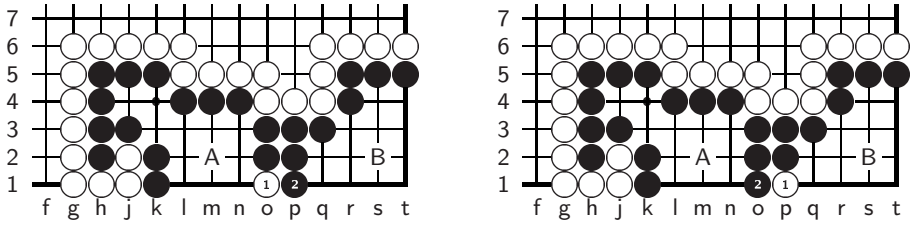


Fig. 2. Single region dividing by using miaipair $\{o1, p1\}$

strategy, Black can divide R into two subregions, A on the left and B on the right.

Assume a region R is divided into two open boundary subregions A and B by a miaipair $M = \{p_A, p_B\}$, such that p_A is adjacent to A and p_B is adjacent to B. Local safety searches are performed for $A \cup M$ and $B \cup M$. The safety search in an open boundary region is similar to the one described in [6], but constrained by the miai strategy outlined above as follows (shown for $A \cup M$).

1. The attacker can select any legal move in $A \cup M$, as long as both points in M are empty.
2. The defender checks whether a forced miai reply exists. If the attacker just occupied either p_A or p_B and the other miai point is still open, the defender immediately replies there.
3. Otherwise, the defender can choose any legal move in $A \cup \{p_A\}$ (but not p_B).

For example, when searching A in Fig. 2, White as the attacker can select any move in A as well as both moves from the miaipair $\{o1, p1\}$. Black can choose the same moves except $p1$. If White plays first at $o1$ or $p1$, Black must take the other. However if Black plays $o1$ first, the miai strategy is fulfilled and conditions need not be checked in the future. The move $p1$ is also removed from White's options.

The basic algorithm to prove that a dividable single region R is safe by using a miaipair-constrained safety solver is shown below. The algorithm takes another parameter S, the set of points (possibly including boundary blocks) previously shown to be safe by using other regions.

1. Use miaipair M to divide region R into two open boundary subregions A and B.
2. Run solver for A and compute new set of safe points: $newS$=$solve(A, \{M\}, S)$.
3. A was proven safe iff $newS \neq S$.
   (a) If $newS \neq S$, then use $newS$ to try to prove subregion B: $S = solve(B, \{M\}, newS)$.
   (b) If $newS = S$, then run solver on B: $newS = solve(B, \{M\}, S)$. If B is safe, then try to use the newly proven boundary blocks of B to prove A again: $S = solve(A, \{M\}, newS)$.

The result can be summarized as follows.

- If both A and B were proven safe, then R and all its boundary blocks are proven safe.
- Otherwise, if exactly one subregion is proven safe, then that region, its surrounding blocks, and the closer miai point are marked as safe. In the example, if only A were proven safe, then A, its boundary blocks, and $p_A = o1$, (a total of 30 points) would be marked as safe.
- If both local searches fail, nothing is proven safe.

For the example in Fig. 2, both sides and therefore the whole region can be proven safe by our system. In Fig. 3, the original large black region (size: 31) is divided into two subregions by miaipair $\{o1, p1\}$. Only the subregion in the right corner can be proven as safe. Its territory is marked by S and the safe boundary block is marked by triangles.

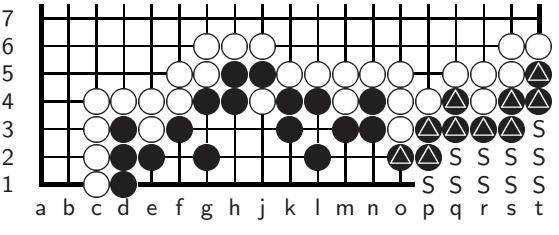


Fig. 3. Part of region is proven safe

3.3 Dividing a Single Region by Multiple Miaipairs

The basic method of Subsection 3.2 is restricted to single miaipairs within a region. Regions with multiple miaipairs can potentially be subdivided in many different ways. The easy case is *independent miaipairs*, where no two pairs are adjacent or overlap.

Figure 4 shows an example. The black region R of size 40 contains two independent miaipairs $P_1 = \{f1, g1\}$ and $P_2 = \{o1, p1\}$ that divide R into three partial regions, A to the left of P_1, B between P_1 and P_2, and C to the right of P_2. Since B is bounded by two miaipairs, both will be passed to the search

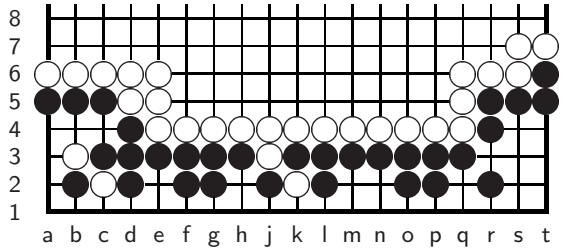


Fig. 4. First case of using multiple miaipairs together to divide a large region

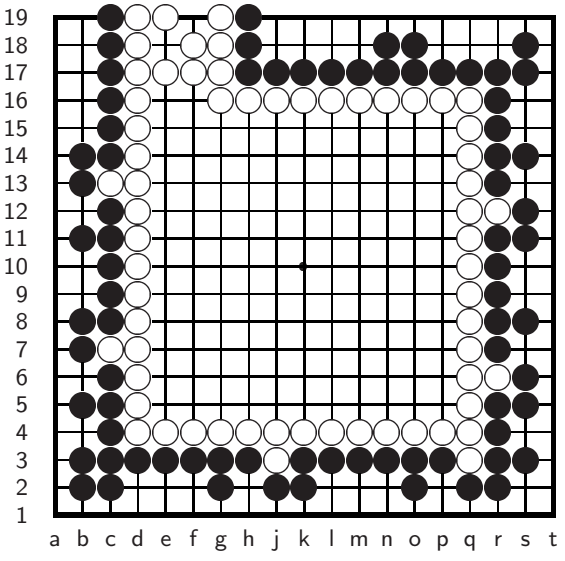


Fig. 5. Using 10 miaipairs to prove safety of a large black region

$solve(B, \{P_1, P_2\}, S)$. In general, a partial region bounded by n independent miaipairs is divided into $n+1$ partial regions that can be searched separately. Every time a partial region is proven safe, S is updated to include the region and its safe boundary. Then, subregion searches continue until no further updates can be made.

Figure 5 shows an extreme case from our test set, a huge black region of size 174. It contains 10 independent miaipairs. No previous search method can prove its safety. Using only a single miaipair at a time, just two small partial regions at the ends can be proven safe. The complete method proves the safety of the whole region.

If miaipairs are adjacent to each other or overlap, they cannot all be used. For example in the right of Fig. 1, the two miaipairs $P_1 = \{d1, e1\}$ and $P_2 = \{e1, f1\}$, cannot divide the region into three subregions. In this case the current implementation first computes clusters of miaipairs which are adjacent or overlapping,

then selects a single pair from each cluster. The selection is originally biased towards miaipairs that minimize the size of the largest subregion, but can be modified by backtracking if subproblems fail. In this example, miaipair P_1 is chosen first to find the largest possible safe area. If the whole region cannot be proven as safe by using P_1, then other miaipairs in this cluster will be tried.

3.4 Dividing a Merged Region

A set of strongly or weakly related regions can be merged into a large region, as described in [5]. After dividing a region, the resulting subregions may need to be merged with their respective strongly or weakly related regions. Figure 6 shows an example from the test set.

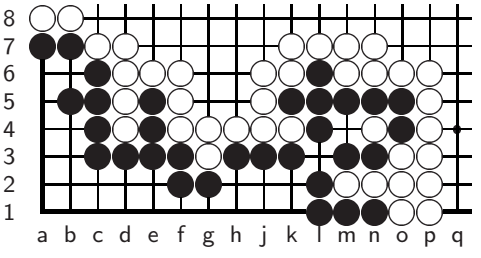


Fig. 6. Dividing a merged region

The original algorithm from [5] merges the regions r_1 at $a1$, r_2 at $l3$, and r_3 at $m4$ into a single region. r_1 contains the only miaipair $\{f1, g1\}$. The current algorithm first splits r_1 into subregions A on the left and B on the right. At this point related regions need to be merged. In the example, A has no related regions and can be tested on its own, but since B is strongly related to subregions r_2 and r_3, the (sub)regions B, r_1, and r_2 are merged into a new partial region for testing.

4 Experimental Results

SAFETY SOLVER 3.0 enhances the previous *SAFETY SOLVER 1.0* and *SAFETY SOLVER 2.0* with the new partial-region proving module.

There are two test sets for experiments to test the performance of *SAFETY SOLVER 3.0*. The first set contains 21 test positions. Each position contains either a large single region or a large merged region [5]. 17 of these test positions are taken from master games. The remaining 4 positions were created by the authors to test extreme cases with many miaipairs. The second test set is the collection of 67 master games introduced in [6]. The test sets are available at: `http://games.cs.ualberta.ca/go/partial`.

All experiments were performed on a Pentium IV/1.7GHz machine with 1 Gb memory. The following abbreviations for the solvers and enhancements are used in the text.

BENSON. Benson's algorithm, as in [3].
STATIC. Static safety-of-territory solver from [6].
SOLVER 1.0. Search-based safety-of-territory solver as described in [5]. It uses regions for board partitioning.
SOLVER 1.0 + P. Solver 1.0 + partial-region proving module.
SOLVER 2.0. Open boundary safety-of-territory solver as described in [6].
SOLVER 3.0. Solver 2.0 + partial-region proving module, the full solver.

4.1 Experiment One: Partial Region Solving

The purpose of this experiment is to test the performance improvements of the partial-region proving module in *SOLVER 1.0*. Since *SOLVER 2.0* uses heuristically computed open boundary zones for board partitioning, many positions in this experiment cannot be recognized. Therefore *SOLVER 2.0* is not compared in this subsection. For all 21 positions, the time limit is set to 200 seconds per search.

The only test position not solved by *SOLVER 1.0 + P* is shown in Fig. 7. The white region (size: 25) can be nicely divided into two small subregions A (size: 11) and B (size: 12) by using miaipair $\{k19, l19\}$. However, neither subregion can be proven safe due to the conservative assumption that no external liberties of boundary blocks may be used to establish safety. For example, when searching subregion A on the left, after move sequence $(B : k19, W : l19)$ White's boundary block at $k18$ is considered to be in atari by the solver because the external liberties at $m18$ and $m19$ may not be used. The situation for proving subregion B is analogous.

For the remaining 20 positions, *SOLVER 1.0 + P* finds at least some safe partial regions. Most of these 20 positions have size larger than 18 points. *SOLVER 1.0* can only prove 4 of them safe within 200 seconds. For a further analysis of the performance improvements, we divide these 20 positions into three groups.

Group 1 contains the 4 test positions that can be proven safe by both *SOLVER 1.0* and *SOLVER 1.0 + P*. Table 1 compares the solution time and number of expanded nodes for both solvers. In all 4 positions, the partial-proving module greatly improves the solver's performance. For example, Fig. 8 shows Position 21 at the top left corner and Position 11 at the bottom right corner. *SOLVER 1.0 + P* is over 61 times faster than *SOLVER 1.0*. However when solving Position 21 (size: 18), *SOLVER 1.0 + P* is only 3.3 times faster. The partial-region proving module first finds the most evenly dividing miaipair $\{r1, s1\}$, then performs

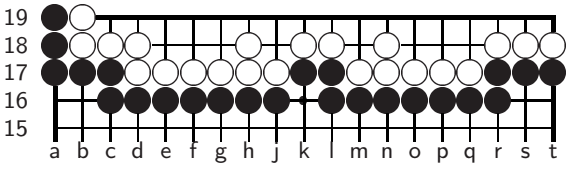


Fig. 7. The only position in set 1 that can not be proven safe

Table 1. Comparison of performance improvements

Position		*SOLVER 1.0*		*SOLVER 1.0 + P*	
Name	Size	Time (Seconds)	Nodes Expanded	Time (Seconds)	Nodes Expanded
No.2	15	8.5	25,993	2.10	1,785
No.8	16	25.55	70,133	1.64	1,752
No.11	18	120.13	236,786	35.7	45,123
No.21	18	156.57	232,332	2.53	3,506

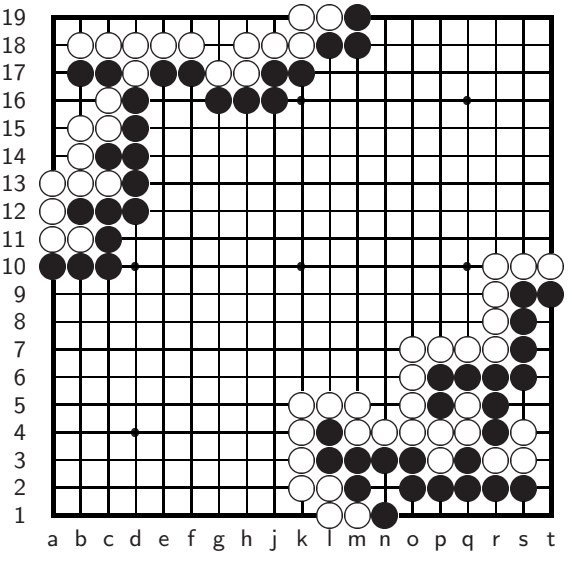


Fig. 8. Examples from group 1

safety searches to prove the whole region safe in 35.7 seconds. Our conclusion is that by using the miaipair $\{b19, c19\}$the division in Position 21 is quite even, each partial region has a similar small size. Therefore each local search is very fast. In contrast, the division in Position 11 is not that even. When using the most evenly dividing miaipair $\{p1, q1\}$, the left and right partial regions have the sizes of 4 and 12. Therefore the local search in the right partial region still requires longer time.

Group 2 contains the 6 test positions that can only be proven partially safe by *SOLVER 1.0 + P*. The top of Fig. 9 shows a real game position from this group. The program cannot prove the whole black region (size: 50) safe. However, by using the miaipair $\{p19, q19\}$ it proves that the partial region S at the top right corner and its boundary blocks (marked by triangles) are safe in 0.47 seconds.

Group 3 contains 10 test positions with either a large single region or a large merged region. *SOLVER 1.0 + P* can prove the whole region safe for every position. The bottom of Fig. 9 shows a real game position from this group. The black merged region contains two subregions r_1 at $b2$ and r_2 at $e3$. The size

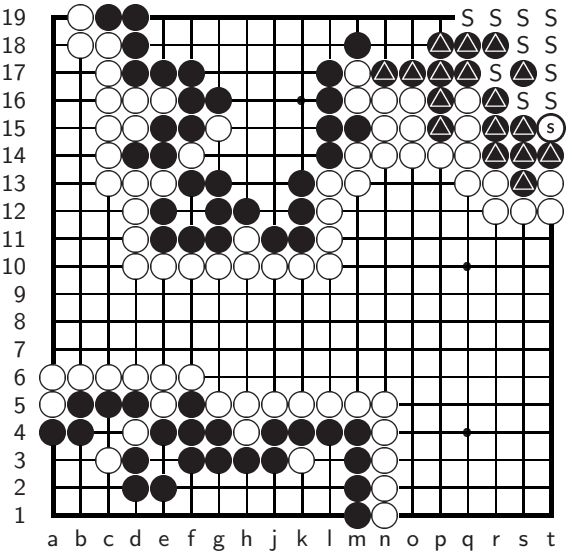


Fig. 9. Examples from group 2 and 3

of the merged region is 28. By using the miaipair $\{d1, e1\}$, *SOLVER 1.0 + P* proves it safe in 72 seconds. A second example from this group is the position shown in Fig. 5, which contains a huge region (size: 174) with multiple miaipairs. *SOLVER 1.0 + P* proves this extreme case safe in 63 seconds.

4.2 Experiment Two: Comparison of Solvers

This experiment compares the performance of solvers *BENSON*, *STATIC*, *SOLVER 1.0*, *SOLVER 2.0*, *SOLVER 1.0 + P* and *SOLVER 3.0* on 67 completed games. The time limit is set to 20 seconds per search. As in [6], each solver computes the proven safe points starting from the end of the game, then backwards every 25 moves. Table 2 shows the total number of proven safe points for all six solvers.

Table 2. Comparison of solvers on 67 games

Game Phases	End-100	End-75	End-50	End-25	End
BENSON	19	63	257	600	2,571
STATIC	106	242	587	1,715	5,584
SOLVER 1.0	234	462	1,138	3,189	10,285
SOLVER 2.0	594	838	1,651	3,653	10,815
SOLVER 1.0 + P	292	540	1,704	3,765	11,299
SOLVER 3.0	606	884	2,179	4,227	11,725

SOLVER 3.0 can prove the most points safe in all game phases. Interestingly, in earlier game phases such as *End - 100* and *End - 75*, the open boundary *SOLVER 2.0* beats *SOLVER 1.0 + P* by a large margin. It seems that in such early stages, there are not enough miaipairs. Thus the open boundary solver is more useful. By *End - 50*, *SOLVER 1.0 + P* has caught up to *SOLVER 2.0*'s performance. The combined *SOLVER 3.0* proves 27% more safe points than *SOLVER 2.0*. In *End - 25* and *End* stages, the improvements of *SOLVER 3.0* are 15% and 8% respectively.

5 Conclusions and Future Work

In this paper we have presented a partial-region safety-proving technique. From the experimental results presented above we may concluded that *SAFETY SOLVER 3.0* enhanced with this technique significantly outperforms previous solvers.

Below we provide two promising ideas for further enhancements.

1. Generalize region splitting techniques. The current technique is limited to adjacent miaipairs. (1a) An extension would be utilizing all possible single splitting points to divide a region. For example, in the left of Fig. 1, all four splitting points at $c1, e1, f1$ and $j1$ could be used in the safety search. (1b) A second extension would be to divide a region by other, larger gaps such as diagonal and one point jumps.
2. The aim of the current safety solver is to prove the safety status of territories. Applying the prover to real game playing and building a quick and strong heuristic safety analyzer for attacking or defending territory is a second interesting topic.

References

1. D.B. Benson. Life in the game of Go. Information Sciences, 10(2),17–29, 1976; Levy, D.N.L. (ed.): Reprinted in Computer Games, Vol. II, pp. 203-213. Springer, New York (1988)
2. Dyer, D.: An eye shape library for computer Go, http://www.andromeda.com/people/ddyer/go/shape-library.html
3. Müller, M.: Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In: Matsubara, H. (ed.) Game Programming Workshop in Japan 1997, Tokyo, Japan, pp. 80–86. Computer Shogi Association (1997)
4. Niu, X., Kishimoto, A., Müller, M.: Recognizing seki in computer Go. 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. 88–103. Springer, Heidelberg (2006)
5. Niu, X., Müller, M.: An improved safety solver for computer Go. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 97–112. Springer, Heidelberg (2006)
6. Niu, X., Müller, M.: An open boundary safety solver in computer Go. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 37–49. Springer, Heidelberg (2007)

7. Vilà, R., Cazenave, T.: When one eye is sufficient: a static classification. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10, pp. 109–124. Kluwer, Dordrecht (2003)
8. van der Werf, E.C.D.: AI techniques for the game of Go. PhD thesis, Maastricht University (2005)
9. Wolf, T., Pratola, M.: A library of eyes in Go, II: Monolithic eyes, In: Games of No Chance 3 (to appear, 2006)

Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength

Rémi Coulom

Université Charles de Gaulle, INRIA SEQUEL, CNRS GRAPPA, Lille, France
Remi.Coulom@univ-lille3.fr

Abstract. Whole-History Rating (WHR) is a new method to estimate the time-varying strengths of players involved in paired comparisons. Like many variations of the Elo rating system, the whole-history approach is based on the dynamic Bradley-Terry model. But, instead of using incremental approximations, WHR directly computes the exact maximum a posteriori over the whole rating history of all players. This additional accuracy comes at a higher computational cost than traditional methods, but computation is still fast enough to be easily applied in real time to large-scale game servers (a new game is added in less than 0.001 second). Experiments demonstrate that, in comparison to Elo, Glicko, TrueSkill, and decayed-history algorithms, WHR produces better predictions.

1 Introduction

Institutions that organize competitive activities, such as sports or games, often rely on ratings systems. Rating systems provide an estimation of the strength of competitors. This strength estimation makes it possible to set up more balanced matches, motivate competitors by providing them with a measurement of their progress, and make predictions about the outcomes of future competitions.

Almost every institution designed its own rating system, so many algorithms exist. The following discussion summarizes the main kinds of rating systems.

Static Rating Systems. Static rating systems do not consider the variation in time of the strengths of players. They are appropriate for rating humans over a short period of time, or for rating computers. An effective method for a static rating system consists in using Bayesian inference with the Bradley-Terry model [1]. But static rating systems are not adapted to situations where players may make significant progress.

Incremental Rating Systems. Incremental rating systems, such as the FIDE rating system [4], Glicko [7], or TrueSkill [8] store a small amount of data for each player (one number indicating strength, and sometimes another indicating uncertainty). After each game, this data is updated for the participants in the game. The rating of the winner is increased, and the rating of the loser is decreased.

Incremental rating systems can handle players of time-varying strength, but do not make optimal use of data. For instance, if two players, A and B, enter the

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 113–124, 2008.

rating system at the same time and play many games against each other, and none against established opponents, then their relative strength will be correctly estimated, but not their strength with respect to the other players. If player A then plays against established opponents, and its rating changes, then the rating of player B should change too. But incremental rating systems would leave B's rating unchanged.

Decayed-history Rating Systems. In order to fix the deficiencies of incremental rating systems, a static rating algorithm may be applied, limited to recent games. This idea may be refined by giving a decaying weight to games, either exponential or linear[1]. With this decay, old games are progressively forgotten, which allows to measure the progress of players.

This decayed-history approach solves some problems of incremental rating systems, but also has some flaws. The main problem is that the decay of game weights generates a very fast increase in the uncertainty of player ratings. This is unlike incremental systems that assume that rating uncertainty grows like the square root of time. With the decayed-history approach, players who stop playing for a while may experience huge jumps in their ratings when they start playing again, and players who play very frequently may have the feeling that their rating is stuck. If players do not all play at the same frequency, there is no good way to tune the speed of the decay.

Accurate Bayesian Inference. An approach that may be more theoretically sound than decayed history consists in using the same model as incremental algorithms, but with fewer approximations. The weakness of algorithms like Glicko and TrueSkill lies in the inaccuracies of representing the probability distribution with just one value and one variance for every player, ignoring covariance. Authors of incremental algorithms already proposed to correct inaccuracies by running several passes of the algorithm forward and backward in time [2,5,7,10]. Edwards [3], with Edo ratings, proposed a method to estimate directly the maximum a posteriori on the exact model.

The WHR algorithm presented in this paper is similar in principle to Edo ratings, although the numerical method is different (Edo uses MM [9], whereas WHR uses the more efficient Newton's method). On his web site, Edwards wrote that "Edo ratings are not particularly suitable to the regular updating of current ratings". Experiments presented in this paper clearly indicate that he underestimated his idea: evaluating ratings of the past more accurately helps to evaluate current ratings: the prediction rate obtained with WHR outperforms decayed history and incremental algorithms. Also, the WHR algorithm allows to update rapidly ratings after the addition of one game, making it suitable for large-scale real-time estimation of ratings.

Paper Outline. Section 2 presents the dynamic Bradley-Terry model, Sect. 3 is the WHR algorithm, and Sect. 4 presents experimental results on data of the KGS Go server.

[1] This idea is often credited to Ken Thompson (for instance, by Sonas [14]).

2 The Dynamic Bradley-Terry Model

This section briefly presents the dynamic Bradley-Terry model [6] that is the basis for the WHR system.

2.1 Notations

- Player number: $i \in \{1, \ldots, N\}$, integer index
- Elo rating of player i at time t: $R_i(t)$, real number.
- γ rating of player i at time t: $\gamma_i(t)$, defined by $\gamma_i(t) = 10^{\frac{R_i(t)}{400}}$.
- Natural rating of player i at time t: $r_i(t) = \ln \gamma_i(t) = R_i(t)\frac{\ln 10}{400}$.

Elo ratings are familiar to chess players, but are on a rather arbitrary and inconvenient scale. Natural ratings will be used most of the time in this paper, because they make calculations easier.

2.2 Bradley-Terry Model

The Bradley-Terry model for paired comparisons gives the probability of winning a game as a function of ratings:

$$P(\text{player } i \text{ beats player } j \text{ at time } t) = \frac{\gamma_i(t)}{\gamma_i(t) + \gamma_j(t)} \ .$$

The Bradley-Terry model may be generalized to handle draws, advantage of playing first, teams, and multi-player games [9]. In this paper, only the simple formulation will be used, but it would be straightforward to generalize the WHR algorithm to those more complex situations.

2.3 Bayesian Inference

The principle of Bayesian Inference consists in computing a probability distribution over player ratings ($\boldsymbol{\gamma}$) from the observation of game results ($\boldsymbol{G}$) by inverting the model thanks to Bayes formula:

$$p(\boldsymbol{\gamma}|\boldsymbol{G}) = \frac{P(\boldsymbol{G}|\boldsymbol{\gamma})p(\boldsymbol{\gamma})}{P(\boldsymbol{G})} \ .$$

In this formula, $p(\boldsymbol{\gamma})$ is a prior distribution over $\boldsymbol{\gamma}$ (lower-case p is used for probability densities), and $P(\boldsymbol{G})$ is a normalizing constant. $P(\boldsymbol{G}|\boldsymbol{\gamma})$ is the Bradley-Terry model described in the previous section. $p(\boldsymbol{\gamma}|\boldsymbol{G})$ is called the posterior distribution of $\boldsymbol{\gamma}$. The value of $\boldsymbol{\gamma}$ that maximizes $p(\boldsymbol{\gamma}|\boldsymbol{G})$ is the maximum a posteriori, and may be used as an estimation of the strengths of players, derived from the observation of their game results.

2.4 Prior

In the dynamic Bradley-Terry model, the prior has two roles. First, a prior probability distribution over the range of ratings is applied. This way, the rating of a player with 100% wins does not go to infinity. Also, a prior controls the variation of the ratings in time, to avoid huge jumps in ratings.

In the dynamic Bradley-Terry model, the prior that controls the variation of ratings in time is a Wiener process:

$$r_i(t_2) - r_i(t_1) \sim \mathcal{N}\big(0, |t_2 - t_1|w^2\big) \ .$$

w is a parameter of the model that indicates the variability of ratings in time. The extreme case of $w = 0$ would mean static ratings.

Some realizations of a Wiener process are plotted on Fig. 1. The Wiener process is a model for Brownian motion, and can be simulated by adding an independent normally-distributed random value at each time step. This means that the variance increases linearly with time, so the confidence interval grows like the square root of time. A Wiener process is said to be memoryless (or Markovian), that is to say, if $t_1 < t_2 < t_3$,

$$p\big(r(t_3)\big|r(t_1), r(t_2)\big) = p\big(r(t_3)\big|r(t_2)\big) \ ,$$
$$p\big(r(t_1)\big|r(t_2), r(t_3)\big) = p\big(r(t_1)\big|r(t_2)\big) \ .$$

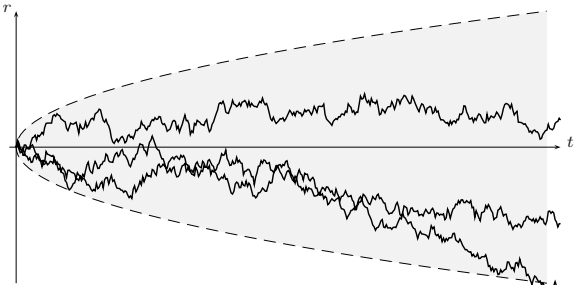


Fig. 1. Three realizations of a Wiener process, with $r(0) = 0$. The dashed line indicates the 95% confidence interval for $p(r(t)|r(0) = 0)$.

3 Algorithm

The WHR algorithm consists in computing, for each player, the $\gamma(t)$ function that maximizes $p(\boldsymbol{\gamma}|\boldsymbol{G})$. Once this maximum a posteriori has been computed, the variance around this maximum is also estimated, which is a way to estimate rating uncertainty.

(a) History of a player (b) optimizing one by one (c) Newton's method

Fig. 2. Newton's method applied to the history of one player. (a) The rating history of a player who has played 4 games is defined by 4 ratings r_1, r_2, r_3, and r_4, at the four times of the four games. (b) Two ratings that are close in time, such as r_2 and r_3, are strongly correlated. So, methods such as MM [9] that optimize parameters one by one, are very inefficient. (c) Since the optimized function is very similar to a quadratic form, Newton's method is extremely efficient.

3.1 Optimization Method

The first step of the WHR algorithm consists in computing the maximum a posteriori for all the $\gamma_i(t)$. Since γ_i are functions of time, this is an infinite-dimensional optimization problem. But it is easy to reduce it to a finite-dimensional problem, since knowing the values of γ at the times of games is sufficient. Rating estimation between two consecutive games may be done by interpolation formulas provided in Appendix C.

Figure 2 illustrates how optimization is performed by Newton's method. Since the Wiener process is Markovian, the Hessian matrix $\left(\frac{\partial^2 \log p}{\partial \boldsymbol{r}^2}\right)$ is tridiagonal, so Newton's method has a cost linear in the number of ratings. Formally, Newton's method consists in updating the rating vector $\boldsymbol{r}$ of one player (the vector of ratings at times when that player played a game) according to this formula:

$$\boldsymbol{r} \leftarrow \boldsymbol{r} - \left(\frac{\partial^2 \log p}{\partial \boldsymbol{r}^2}\right)^{-1} \frac{\partial \log p}{\partial \boldsymbol{r}}$$

The complete details of how to perform this update are provided in Appendices A and B. Since p is very similar to a Gaussian, $\log p$ is very similar to a quadratic form, so only one iteration of Newton's method is sufficient to obtain a very good value of $\boldsymbol{r}$. The overall optimization algorithm consists in applying this Newton update to every player in turn, and iterate until convergence.

3.2 Incremental Updates

The global optimization algorithm described in the previous section may take a few minutes to converge on a big database of games (see Sect. 4). So, it may be too slow to restart the algorithm from scratch in order to estimate new ratings when one new game is added to the database.

In order to let WHR work in real time when one new game is added, a simple solution consists in keeping the rating estimations obtained before the addition, and applying Newton's method once to every player of this game. This is orders

of magnitude faster, although a little less accurate. If computation time allows, more iterations of Newton's method may be applied from time to time. For instance, in experiments of incremental WHR described in the next section, one iteration was run on every player every 1000 games.

3.3 Estimating Rating Uncertainty

Once the maximum a posteriori has been found, rating uncertainty may be estimated. Since the posterior probability is similar to a Gaussian, its covariance matrix may be approximated by the opposite of the inverse of the Hessian.

In practice it is not possible to compute the whole covariance matrix for all the parameters at the same time. But rating uncertainty of one player can be estimated by considering the Hessian of the ratings of this player only, assuming opponents have a fixed rating equal to the maximum a posteriori. The detailed formulas for this operation can be found in Appendix B.2.

This numerical algorithm is extremely similar to a technique developed by physicists to estimate functions from noisy observations [11,12].

4 Experiments in the Game of Go

4.1 Speed of Convergence

The WHR algorithm was tested on the database of games of the KGS Go server. This database contains all rated games since 2000 and until October, 2007. It contains 213,426 players, and 10.8 million games. Computations were performed on an Intel Core2 Duo at 2.4 GHz (using only one thread), and took about 7 minutes for 200 iterations. The prior was one virtual win and one virtual loss against a virtual player of rating zero, on the day of the first game. The Wiener process had a variance of $w^2 = 60$ Elo^2 per day (see Table 1, optimal parameters).

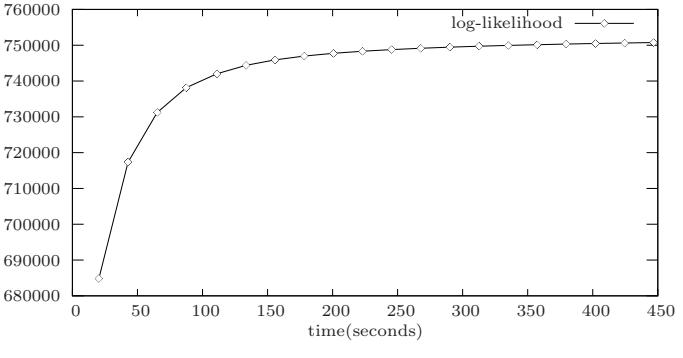


Fig. 3. Speed of optimization. One point is plotted every 10 iterations. Log-likelihood is the difference between the current log-likelihood and the initial log-likelihood, when all ratings were set to zero.

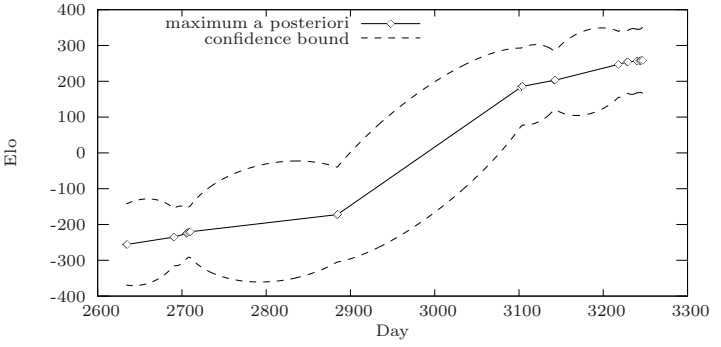


Fig. 4. Whole-History Rating estimation of player Crazy Stone. Dots indicate days when games were played.

Figure 3 shows the convergence speed of the log-likelihood. Figure 4 presents the result of Crazy Stone, a program that made considerable progress in the year 2007. This figure shows that rating uncertainty increases slowly during long periods of inactivity.

4.2 Prediction Ability

The prediction ability of WHR was compared experimentally with the basic Elo algorithm [4], TrueSkill [8], Glicko [7], Bayeselo [1], and decayed history. Bayeselo may be considered as a special case of WHR, with $w^2 = 0$, or a special case of decayed history, with an infinitely long decay. Decayed history was implemented with an exponential decay, that is to say each game was weighted with a coefficient $e^{(t-t_0)/\tau}$, where τ is a parameter of the algorithm.

Bayeselo, WHR, and decayed history all used the same scheme for incremental update of ratings: after each addition of a game, the ratings of the two participants were optimized with one step of Newton's method. Before each prediction of the outcome of a game, the ratings of the two participants were optimized, too. Every 1000 games, the ratings of all participants were optimized, one by one.

The method to compare algorithms consisted in measuring their prediction rates over a database of games. The prediction rate is the proportion of games of which the most likely winner was predicted correctly (when two players have the same rating, this counts as 0.5 correct prediction). Parameters of the algorithm were first tuned to optimize the prediction rate over a training database, then the prediction rate was measured on a different test database.

The training set was made of the 726,648 rated even games with komi 6.5 played on KGS between 2000-11-07 and 2005-05-20. The test set consisted of the 2,331,757 rated even games with komi 6.5 played between 2005-05-21 and 2007-10-01. The time resolution of the database is one day, so if a player played several games in one day, they were considered as simultaneous.

Table 1. Prediction performance of some rating algorithms. Prior = 1 means one virtual win and one virtual loss against a player of rating zero. 95% confidence of superiority is obtained with a difference in prediction rate of 0.163% in the training set, and 0.091% in the test set. Since the same data was used to measure the performances of all algorithms, there is some variance reduction, so differences may be more significant. Time was measured on the training set.

Algorithm	Time	Training	Test	Optimal parameters
Elo	0.41 s	56.001%	55.121%	$k = 20$
Glicko	0.73 s	56.184%	55.522%	$\sigma_0 = 150$ Elo, $w^2 = 20$ Elo^2/day
TrueSkill	0.40 s	56.212%	55.536%	$\beta^2 = 1$, $\sigma_0^2 = 0.5$, $w^2 = 0.000975$/game
Bayeselo	88.66 s	56.216%	55.671%	prior = 1
Decayed history	89.86 s	56.260%	55.698%	prior = 1, $\tau = 400$ days
WHR	252.00 s	56.356%	55.793%	prior = 1.2, $w^2 = 14$ Elo^2/day

Results are summarized in Table 1. It shows that WHR significantly outperforms the other algorithms. Algorithms that remember all the game results outperform the fast incremental methods.

Performance on the test set is inferior to performance on the training set. This probably cannot be explained by overfitting alone. Because these two sets of games correspond to different periods of time, they do not have the same statistical properties. It may be that the KGS rating system improved, and since matches are automatically balanced by the server, recent games are more balanced, so they are more difficult to predict.

A remarkable aspect of these results is that parameters that optimize prediction rate give a very low variance to the Wiener process. The static Bayeselo algorithm even outperformed incremental algorithms on the test set. This is surprising, because many players on KGS are beginners, and made very big progress during the 2.5 years of the test set. The high number of experienced players probably outweighed beginners. This is, by the way, an important inaccuracy in the dynamic Bradley Terry model: it does not take those different abilities to make progress into consideration.

5 Conclusion

WHR is a new rating algorithm that directly computes player ratings as a function of time. From the above results we may conclude that (1) it is computationally more costly than incremental and decayed-history algorithms, but (2) more accurate, and fast enough to be applied in real time to large-scale game servers.

A research direction would be to apply WHR to more data sets, and compare it empirically to more alternative rating algorithms, such as Edo, and TrueSkill Through Time (TTT). It is likely that WHR would not outperform Edo and TTT in terms of prediction rate, since their models are almost identical. But the main difference may be in computation time. Previous experiments with Edo and TTT were run with a time resolution of one year, whereas WHR operates

with a time resolution of one day. Such a short time period between ratings induces a very strong correlation between parameters, which Newton's method may handle more efficiently than MM or approximate message passing.

A second research direction would be to improve the model. An efficient application of WHR to Go data would require some refinements of the dynamic Bradley-Terry model, which the KGS rating algorithm [13] already has. In particular, it should be able to

- Take handicap, komi, and time control into consideration,
- Deal with outliers,
- Handle the fact that beginners make faster progress than experts.

Acknowledgments. I thank William Shubert for providing KGS data. I am also very grateful to the referees, whose remarks helped to improve this paper considerably.

References

1. Coulom, R.: Bayeselo (2005), http://remi.coulom.free.fr/Bayesian-Elo/
2. Dangauthier, P., Herbrich, R., Minka, T., Graepel, T.: TrueSkill through time: Revisiting the history of chess. In: Platt, J.C., Koller, D., Singer, Y., Roweis, S. (eds.) Advances in Neural Information Processing Systems 20, Vancouver, Canada, MIT Press, Cambridge (2007)
3. Edwards, R.: Edo historical chess ratings (2004), http://members.shaw.ca/edo1/
4. Elo, A.E.: The Rating of Chessplayers, Past and Present. Arco Publishing, New York (1978)
5. Fahrmeir, L., Tutz, G.: Dynamic stochastic models for time-dependent ordered paired comparison systems. Journal of the American Statistical Association 89(428), 1438–1449 (1994)
6. Glickman, M.E.: Paired Comparison Model with Time-Varying Parameters. PhD thesis, Harvard University, Cambridge, Massachusetts (1993)
7. Glickman, M.E.: Parameter estimation in large dynamic paired comparison experiments. Applied Statistics 48(33), 377–394 (1999)
8. Herbrich, R., Graepel, T.: TrueSkill^{TM}: A Bayesian skill rating system. Technical Report MSR-TR-2006-80, Microsoft Research (2006)
9. Hunter, D.R.: MM algorithms for generalized Bradley-Terry models. The Annals of Statistics 32(1), 384–406 (2004)
10. Knorr-Held, L.: Dynamic rating of sports teams. The Statistician 49(2), 261–276 (2000)
11. Rybicki, G.B., Hummer, D.G.: An accelerated lambda iteration method for multilevel radiative transfer. Astronomy and Astrophysics 245(1), 171–181 (1991)
12. Rybicki, G.B., Press, W.H.: Interpolation, realization, and reconstruction of noisy, irregularly sampled data. The Astrophysical Journal 398(1), 169–176 (1992)
13. Shubert, W.M.: Details of the KGS rank system (2007), http://www.gokgs.com/help/rmath.html
14. Sonas, J.: Chessmetrics (2005), http://db.chessmetrics.com/CM2/Formulas.asp

A Gradient and Hessian Matrix for One Player

A.1 Terms of the Bradley-Terry Model

The result of one game G_j may be written as:

$$P(G_j) = \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}} \ ,$$

where A_{ij}, B_{ij}, C_{ij}, and D_{ij} are constants that do not depend on γ_i.

$W(i)$ is the set of games that i won, and $L(i)$ the set of games that i lost. $r = \ln \gamma$, so $dr = \frac{d\gamma}{\gamma}$.

$$\ln P = \sum_{j \in W(i)} \ln(A_{ij}\gamma_i) + \sum_{j \in L(i)} \ln(B_{ij}) - \sum_j \ln(C_{ij}\gamma_i + D_{ij})$$

$$\frac{\partial \ln P}{\partial r_i} = |W(i)| - \gamma_i \sum_j \frac{C_{ij}}{C_{ij}\gamma_i + D_{ij}}$$

$$\begin{aligned}\frac{\partial^2 \ln P}{\partial r_i^2} &= -\gamma_i \left(\sum_j \frac{-C_{ij}^2 \gamma_i}{(C_{ij}\gamma_i + D_{ij})^2} + \frac{C_{ij}}{C_{ij}\gamma_i + D_{ij}} \right) \\ &= -\gamma_i \sum_j \frac{C_{ij} D_{ij}}{(C_{ij}\gamma_i + D_{ij})^2}\end{aligned}$$

A.2 Terms of the Wiener Prior

These terms are added to the Hessian for every pair of consecutive ratings, with $\sigma^2 = |t_2 - t_1| w^2$.

$$\begin{aligned}\frac{\partial \ln p}{\partial r_i(t_1)} &= -\frac{r_i(t_1) - r_i(t_2)}{\sigma^2} \\ \frac{\partial^2 \ln p}{\partial r_i(t_1)^2} &= -\frac{1}{\sigma^2} \\ \frac{\partial^2 \ln p}{\partial r_i(t_1) \partial r_i(t_2)} &= \frac{1}{\sigma^2}\end{aligned}$$

B LU Decomposition of the Hessian Matrix

$$H = (h_{ij}) = LU = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ a_2 & 1 & 0 & \dots & 0 \\ 0 & a_3 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & a_n & 1 \end{pmatrix} \begin{pmatrix} d_1 & b_1 & 0 & \dots & 0 \\ 0 & d_2 & b_2 & \dots & 0 \\ 0 & 0 & d_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & b_{n-1} \\ 0 & 0 & 0 & \dots & d_n \end{pmatrix}$$

Algorithm ($i \geq 2$):

$$\begin{aligned} d_1 &= h_{11} \\ b_1 &= h_{12} \\ a_i &= h_{ii-1}/d_{i-1} \\ d_i &= h_{ii} - a_i b_{i-1} \\ b_i &= h_{ii+1} \ . \end{aligned}$$

B.1 Computing $H^{-1}G$

The problem is to find vector X so that $LUX = G$. The first step of the algorithm consists in finding Y so that $LY = G$ ($i \geq 2$):

$$\begin{aligned} y_1 &= g_1 \\ y_i &= g_i - a_i y_{i-1} \ . \end{aligned}$$

Then, find X, so that $UX = Y$ ($i < n$):

$$\begin{aligned} x_n &= y_n/d_n \\ x_i &= (y_i - b_i x_{i+1})/d_i \ . \end{aligned}$$

Since $h_{ii+1} > 0$ and $h_{ii} < -(h_{ii-1} + h_{ii+1})$, $-1 < a_i < 0$ and $d_i < -h_{ii+1}$, so this algorithm has no division by zero. In order to ensure numerical stability, 0.001 is subtracted to all diagonal elements of H.

B.2 Computing Diagonal and Sub-diagonal Terms of H^{-1}

The covariance matrix for the full history of one player is approximated by

$$\Sigma = -H^{-1} = \begin{pmatrix} \sigma_1^2 & \sigma_{12} & \dots & \sigma_{1n} \\ \sigma_{12} & \sigma_2^2 & \dots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{1n} & \sigma_{2n} & \dots & \sigma_n^2 \end{pmatrix} \ .$$

In order to compute the confidence envelope around rating histories, only the diagonal and sub-diagonal terms of the reverse of H are needed. This can be done in cost linear in n [11]. The trick consists in doing the UL decomposition as well as the LU decomposition:

$$H = (h_{ij}) = U'L' \begin{pmatrix} 1 & a'_1 & 0 & \dots & 0 \\ 0 & 1 & a'_2 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & a'_{n-1} \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \begin{pmatrix} d'_1 & 0 & 0 & \dots & 0 \\ b'_2 & d'_2 & 0 & \dots & 0 \\ 0 & b'_3 & d'_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & b'_n & d'_n \end{pmatrix} \ .$$

Algorithm ($i < n$):

$$\begin{aligned} d'_n &= h_{nn} \\ b'_n &= h_{nn-1} \\ a'_i &= h_{ii+1}/d'_{i+1} \\ d'_i &= h_{ii} - a'_i b'_{i+1} \\ b'_i &= h_{ii-1} \ . \end{aligned}$$

In order to solve $HX = G$, first solve $U'Y' = G$ ($i < n$):

$$\begin{aligned} y'_n &= g_n \\ y'_i &= g_i - a'_i y'_{i+1} \ . \end{aligned}$$

Then, solve $L'X = Y'$ ($i > 1$):

$$\begin{aligned} x_1 &= y'_1/d'_1 \\ x_i &= (y'_i - b'_i x_{i-1})/d'_i \ . \end{aligned}$$

The i-th term of the diagonal of the inverse of H, σ_i^2, is x_i, computed with $g_j = \delta_{ij}$. It can be computed by using the two decompositions at the same time:

$$\begin{aligned} x_i &= (y_i - b_i x_{i+1})/d_i \\ x_{i+1} &= (y'_{i+1} - b'_{i+1} x_i)/d'_{i+1} \ . \end{aligned}$$

Since $y_i = 1$, and $y'_{i+1} = 0$, we get ($i < n$):

$$\begin{aligned} \sigma_i^2 &= -x_i = d'_{i+1}/(b_i b'_{i+1} - d_i d'_{i+1}) \\ \sigma_n^2 &= -1/d_n \ . \end{aligned}$$

Sub-diagonal elements may be computed with

$$\sigma_{ii-1} = -a_i \sigma_i^2 \ .$$

C Interpolation Formulas

The mean, μ, and variance, σ^2, of a Wiener process at time t may be interpolated between two times t_1 and t_2, assuming that the means at t_1 and t_2 are μ_1 and μ_2, and the covariance matrix is $\Sigma = \left(\begin{smallmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{smallmatrix}\right)$, with the following formulas:

$$\mu = \frac{\mu_1(t_2 - t) + \mu_2(t - t_1)}{t_2 - t_1} \ ,$$

$$\sigma^2 = \frac{(t_2 - t)(t - t_1)}{t_2 - t_1} w^2 + \frac{(t_2 - t)^2 \sigma_1^2 + 2(t_2 - t)(t - t_1)\sigma_{12} + (t - t_1)^2 \sigma_2^2}{(t_2 - t_1)^2} \ .$$

Frequency Distribution of Contextual Patterns in the Game of Go

Zhiqing Liu, Qing Dou, and Benjie Lu

BUPT-JD Computer Go Research Institute,
Beijing University of Posts and Telecommunications,
Beijing, China, 100876
zhiqing.liu@gmail.com

Abstract. In this paper, we present two statistical experiments on the frequency distribution of patterns in the game of Go. In these experiments, we extract contextual patterns of Go as spatial combinations of moves. An analysis of a collection of 9447 professional game records of Go shows that the frequency distribution of contextual patterns in professional games displays a Mandelbrot fit to Zipf's law. Additionally, we show that the Zipfian frequency distribution of Go patterns in professional games is deliberate by rejecting the null hypothesis that the frequency distribution of patterns in random games exhibits a Zipfian frequency distribution.

1 Introduction

Computer games has been a part of the core of artificial intelligence since it became a field of study; and the game of Go is one of its grand challenges [12]. The purpose of this paper is to investigate the frequency distribution of patterns in the game of Go. More, specifically, this paper is to investigate whether the statistical distribution of frequency of occurrence of Go patterns exhibits certain regularities as in the case of natural language.

One of the most obvious universal features in natural languages is the statistical distribution of word frequency of occurrence, which exhibits a prominent regularity. If words in a corpus of sample texts are sorted decreasingly by their frequency of occurrence, the frequency of a word, denoted as f, is a power law function of its rank, denoted as r, in the form of $f \sim 1/r^e$ with the exponent e close to unity [17]. This regularity in natural languages was first observed in the English language by Harvard linguistics professor George Kingsley Zipf, and is now commonly referred to as Zipf's law. Mandelbrot has extended Zipf's law into the more general form of $f = P(r + \rho)^B$ where P, B, and ρ are constants [10].

Two primary contributions of this paper are the following.

1. Patterns extracted from professional Go game records as spatial combinations of moves are shown statistically to exhibit a Zipfian frequency distribution.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 125–134, 2008.

2. Because random sources always generate statistically consistent results [11], frequency distribution from a random source can be regarded as a null hypothesis. In this sense, the Zipfian frequency distribution of Go patterns is shown to be deliberate and not accidental by a rejection of the null hypothesis.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction of necessary background of the game and reviews related works on computer Go. Following that, two statistical experiments are presented. Section 3 presents frequency distribution of patterns extracted automatically from the same collection of professional games, which has a good Mandelbrot fit to Zipf's law. Section 4 analyzes the frequency distribution of patterns similarly extracted from a collection of randomly generated game records such that the null hypothesis is rejected. Section 5 concludes this paper with a summary of results and a discussion of future works.

2 The Game of Go and Computer Go

Some essential knowledge of the game is necessary. Go is a zero-sum and perfect-information board game of two players, who play the game by alternately placing stones of her color (black or white) on an empty crossing on a board of the game, which consists of a 19-by-19 grid. We, following the tradition of Chess, shall refer to a stone placed on the game board as a move. A game record of Go is, in addition to some supplemental information such as identification of the players and the result, just a sequence of moves, each of which is denoted as a two-dimension coordinate representing the position of the game board on which a stone is placed by the move. In general, the stone of a move can be placed at any empty crossing on the game board, subject to certain rules of the game such as the rule of no-suicide and the rule of ko.

Computer Go has been studied since 1969, when the first paper on this subject was published by Zobrist [18]. Many important progresses on computer Go have since been made, and some of the most influential works include [3]. However, all these works are still elusive with respect to the ultimate goal of defeating top-level human players in Go. The reasons are well recognized: key computer techniques for successful play of other games such as Chess do not apply to Go. One of the key techniques is an accurate static evaluation of the game board integrated with an efficient search of the game tree. However, this technique is almost useless in computer Go because of the following two complications of the game.

1. Go has a significantly larger search space than other games due to its much larger branching factor.
2. An accurate and static evaluation of the game board is not tractable [13].

Patterns are extensively used by both human and computer Go players. A pattern in Go is a well established sequence of moves that can be accepted by both players. It is believed in the Go community that patterns, which define relationships among moves, are of much more importance than the individual moves

themselves [7]. In other words, it is relationships among individual moves that effectively determine a player's a player's skills of Go playing. Effective recognition of patterns and their competent use are crucial for both human players as well as computer programs. To this end, dictionaries of patterns at different stages of the game have been compiled, and pattern databases of various forms are used by almost all competent Go programs. The patterns used are either translated from pattern dictionaries directly, acquired through machine learning [2], generated by enumeration with urgency determined by static analysis [6], by human expertise [15], or by reinforcement learning [5], or extracted from game records statistically [9,14].

Although occasionally used in computer Go, statistical methods have not been used for investigation of frequency distribution of moves and patterns in Go. Besides for pattern extraction as reviewed above, statistical methods are used mostly in Monte-Carlo game evaluation and move generation in computer Go [4]. Monte-Carlo evaluation in computer games was formalized in a framework by Abramson [1], in which the expected outcome of a game position was estimated to be the average of the outcomes of N random games from the position given. In this paper we analyze the pattern frequency distribution of random games in order to study and reject the null hypothesis. To this end, we generate a large number of random games as in Monte-Carlo Go.

3 Frequency Distribution of Contextual Patterns in Professional Games

Our first statistical experiment is to investigate the frequency distribution of patterns in a collection of game records of professional players. The professional game collection used is provided by Yu Ping, the Chinese six-dan professional Go player, who has used the collection in his study of Go. The study is completely unrelated to the work reported in this paper. The collection has 9447 unique professional game records, consisting of 2,040,888 individual moves in total.

Patterns used in our experiments are not derived from pattern dictionaries, because dictionary patterns have different physical properties such as shape, area, and size, making study of their frequency of occurrence meaningless. Instead, we specify a fixed area of the game board serving as the region in which patterns are defined. We shall refer to such defined patterns as contextual patterns. The fixed area is a 5-by-5 square, centered by the stone of the current move. Similar results occur for patterns extracted from larger squares such as 7-by-7 and 9-by-9. When the stone is played close to an edge or a corner of the game board, the region for the pattern definition is naturally reduced to its remaining part. In our definition of patterns and in the subsequent study of their frequency of occurrence, equivalent shapes are combined into one single pattern by elimination of symmetry properties of the shapes. Shapes are considered to be equivalent if they are the same with respect to rotation and/or flipping. Shapes are also considered to be equivalent with respect to color switch.

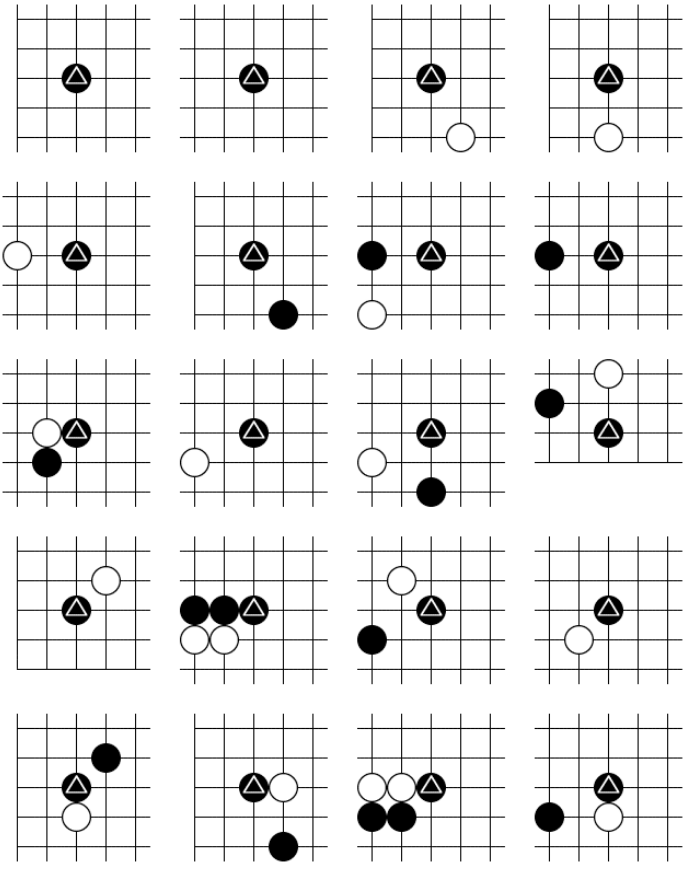


Fig. 1. Top twenty contextual patterns with highest frequency of occurrence in the professional games

Given the above definitions of a pattern and a shape, we are now ready to investigate contextual patterns and their frequency of occurrence to be extracted automatically from game records, because a pattern occurs with each move played. A scan of the collection of professional game records extracts 980,771 unique contextual patterns as defined above. The top twenty patterns with the highest frequency of occurrence are shown in Fig. 1. It is obvious when having some knowledge of Go that all of these patterns are important and are played frequently by professional players. For example, the most frequent pattern shows playing a stone on a third line of the game board, an important move often played at early stages of the game.

Table 1 shows, for the top twenty high-frequency contextual patterns, their numbers of frequency of occurrence (f), orders (r) sorted based on frequency of occurrence, percentages of frequency of occurrence, and products of f and r. The products of f and r appear to be around the constant 50000.

Table 1. Numbers of frequency of occurrence (f), orders (r), percentages of frequency, and the products of $f \times r$ of top twenty high-frequency contextual patterns in the professional games

Order (r)	Frequency (f)	Percentage	$f \times r$
1	39885	1.955%	39885
2	36599	1.792%	73198
3	24091	1.181%	72273
4	12961	0.635%	51844
5	11955	0.586%	59775
6	10049	0.493%	60294
7	5502	0.270%	38514
8	5045	0.247%	40360
9	4962	0.243%	44658
10	4707	0.231%	47070
11	4593	0.225%	50523
12	4143	0.203%	49716
13	3775	0.185%	49075
14	3512	0.172%	49168
15	3385	0.166%	50775
16	3308	0.162%	52928
17	3122	0.153%	53074
18	2806	0.138%	50508
19	2725	0.134%	51775
20	2627	0.129%	52540

Figure 2 shows in a diagram the Zipf curve of the pattern frequency distribution in the collection of professional game records and a Mandelbrot fit to the Zipf's curve. The parameters of the Mandelbrot fitting curve are $P = 50000$, $B = 0.91$, and $\rho = 0.5$, making it almost a straight line, fitting Zipf's law very well.

In summary, frequency of occurrence of 5-by-5 contextual patterns extracted from the collection of professional game records clearly exhibits Zipfian distribution. Additionally, high-frequency patterns are all important and are commonly played by professional players.

4 Frequency Distribution of Contextual Patterns in Random Game Records

Our second statistical experiment is to investigate the frequency distribution of patterns similarly extracted from a collection of randomly generated game records. This experiment is inspired by the works on frequency distribution of random text [8], and its purpose is to reject the null hypothesis that patterns in random games exhibit a Zipfian frequency distribution. The random games used in this experiment are generated through self-play by a random computer

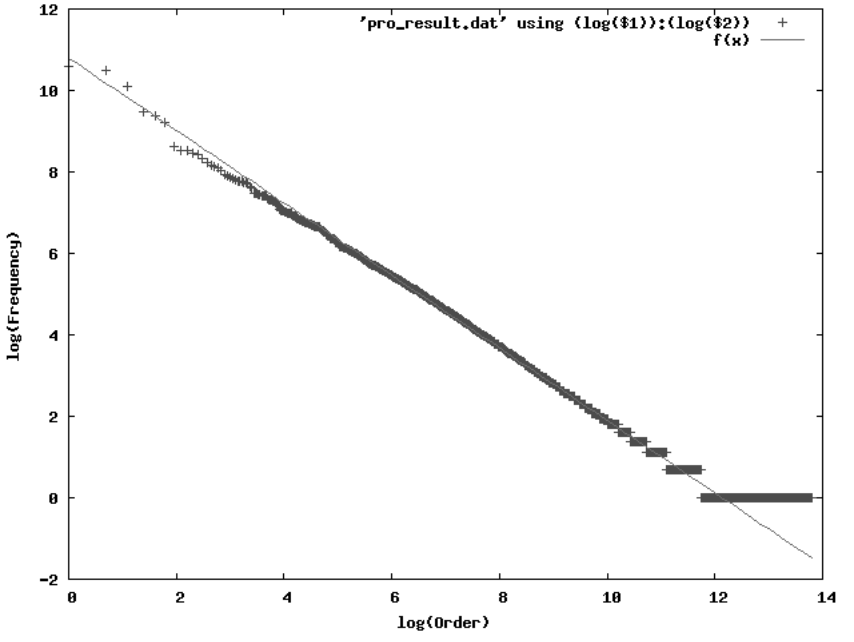


Fig. 2. Mandelbrot fit to a pattern frequency distribution in the professional games. The dots represent the relationships between frequency and order of patterns on a log-log scale, and the line represents the curve of the Mandelbrot formula $f = P(r + \rho)^B$ where $P = 50000$, $B = 0.91$, and $\rho = 0.5$.

Go player. Similar to random players commonly used in Monte-Carlo Go [5], our random computer Go player is assumed to have very little knowledge of the game, and plays completely randomly from all legal moves on the board, excluding only those that are its own eyes.[1] During such a self-play game, the random player keeps on playing randomly until no valid move is available and the game is then scored using the Chinese rules. This random game collection has 3553 records with 1,629,069 individual moves in total. The mean of number of moves in each of the random games is 458.5.

As in the previous experiment, the same definition of 5-by-5 contextual patterns with symmetry elimination is used in this experiment. A scan of the collection of random games extracts 474,117 unique contextual patterns. The top twenty contextual patterns with highest frequency of occurrence are shown in Fig. 3.

Below we give three characteristics of the patterns.

1. All of the contextual patterns have either one stone or two stones.
2. The contextual patterns seen as a set, denoted by $S1$ (see Fig. 1), is significantly different from the set of contextual patterns obtained in the second

[1] In general, an eye is an empty crossing on the game board that is surrounded by stones of one color.

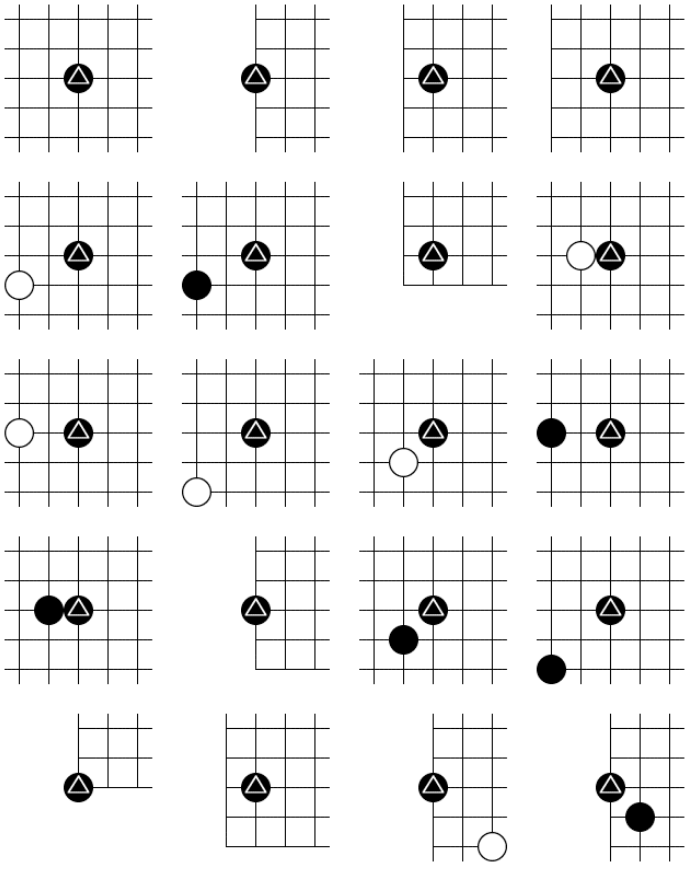


Fig. 3. Top twenty contextual patterns with highest frequency of occurrence in the random games

experiment, denoted by $S2$ (see Fig. 3). They only have six patterns in common.

3. With some knowledge of Go it is obvious that the patterns in $S1$ and $S2$ typically are not important and appear rarely in professional games. For example, the second most frequent pattern in random games shows a stone played on a first line with an empty surrounding area, reflecting a very ineffective move at early stages of the game.

Fig. 4 shows in a diagram the Zipf curve of contextual pattern frequency distribution in the random games. It is evident from the diagram that the contextual pattern frequency distribution does not fit Zipf's law well. There are a number of irregularities. First of all, the curve is far from a straight line as required in Zipf's law. More importantly, the frequency of low-order patterns in random games is significantly higher than what is required by Zipf's law.

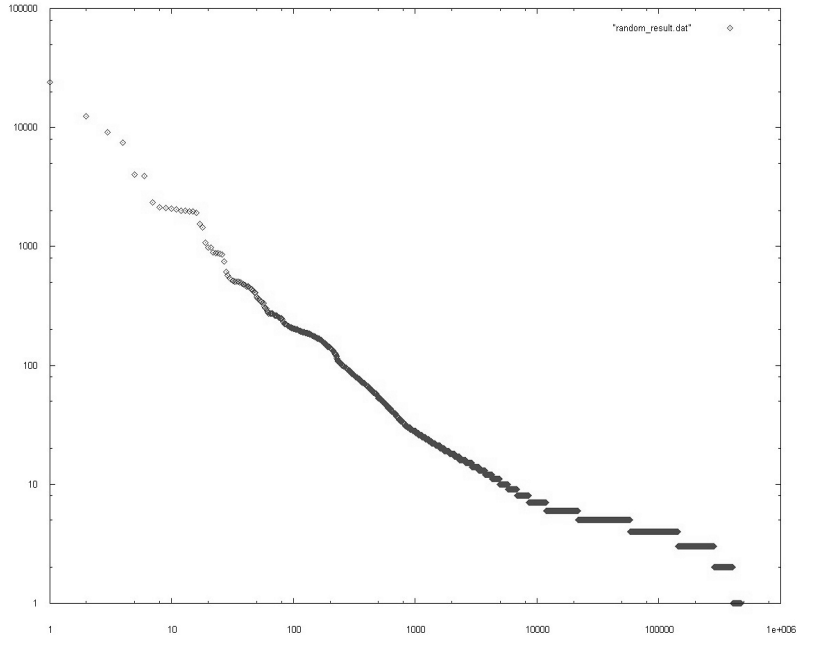


Fig. 4. Contextual pattern frequency distribution in the random games. The dots represent the relationships between frequency and order of patterns on a log-log scale.

5 Discussion and Conclusion

From the two statistical experiments presented above we may draw two conclusions. The primary conclusion is that the game of Go, at the level of contextual pattern, shows clear structures that resemble those of natural languages at the lexical level, because both exhibit Zipfian frequency distribution. However, the result of the Zipfian frequency distribution of Go contextual patterns in professional games is stronger than that of the natural language words in a corpus. The reason is straightforward, unlike the case of random texts, contextual patterns of randomly generated Go games do not exhibit a Zipfian frequency distribution. This shows that the Zifpian frequency distribution of Go patterns in professional games is deliberate by rejecting the null hypothesis.

The second conclusion is that the Zipfian pattern frequency distribution in the game of Go has important consequences in practical work. The most prevailing consequence is data sparseness because of the Zipfian nature of the frequency distribution. Regardless of the size of a game collection, most of the patterns occurring in the games have a very low frequency and a small set of high-frequency patterns constitutes a large majority of occurrence in the game collection. Data sparseness may cause many problems. We mention two of them. First, precise usage of a pattern may not be adequately determined unless the pattern has a reasonably large number of occurrences. As an example in lexicology, Sinclair

suggested that at least 20 instances of an unambiguous word need to be inspected in order to obtain an idea of its behavior [16]. Second, data sparseness dictates that it is statistically impossible to sample all patterns in Go, and it is almost certain that new patterns will be encountered in a new game. This may make statistics-based pattern-analysis methods ineffective, because they violate the randomness assumption, which is at the core of statistical modeling.

Acknowledgments. The author is indebted to Professor Jintong Lin, the President of Beijing University of Posts and Telecommunications for helping establish the BUPT-JD Research Institute of Computer Go, which makes this work possible. The author thanks Yu Ping for insightful discussions of the game and for providing the game records of professional players used in this paper.

References

1. Abramson, B.: Expected-outcome: a general model of static evaluation. IEEE Transactions on PAMI 12, 182–193 (1990)
2. Abramson, M., Harry, W.: A distributed reinforcement learning approach to pattern inference in Go. In: International Conference on Machine Learning Applications, Los Angeles, CA (2003)
3. Benson, D.B.: Life in the game of Go. Information Sciences, 10(2), 17-29, 1976; Levy D.N.L., (ed.). Reprinted in Computer Games Vol. II pp. 203-213. Springer, New York (1988)
4. Bouzy, B., Helmstetter, B.: Monte-Carlo Go Developments. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games, Many Games, Many Challenges, pp. 159–174 (2003)
5. Bouzy, B., Chaslot, G.: Monte-Carlo Go reinforcement learning experiments. In: IEEE 2006 Symposium on Computational Intelligence in Games, Reno, USA (2006)
6. Cazenave, T.: Generation of patterns with external conditions for the game of Go. In: Advances in Computer Games Conference, Paderborn (1999)
7. Cho, C.: Go: A Complete Introduction to the Game. Kiseido Publishing Co (1997)
8. Li, W.: Random texts exhibit Zipf's-law-like word frequency distribution. IEEE Transactions on Information Theory 38(6), 1842–1845 (1992)
9. Liu, Z., Dou, Q.: Automatic Pattern Acquisition from game Records in Go. Journal of China Universities of Posts and Telecommunications 14(1), 100–105 (2007)
10. Mandelbrot, B.B.: Simple games of strategy occurring in communication through natural languages. In Symposium on Statistical Methods in Communication Engineering (1954); Also appeared in Transactions of IRE (professional groups on information theory), 3, 124-137 (1954)
11. Miller, G.A., Chomsky, N.: Finitary models of language users. In: Luce, R.D., Bush, R., Galanter, E. (eds.) Handbook of Mathematical Psychology, vol. 2. Wiley, New York (1963)
12. Müller, M.: Computer Go. Artificial Intelligence 134(1-2), 145–179 (2002)
13. Müller, M.: Position Evaluation in Computer Go. ICGA Journal 25(4), 219–228 (2002)
14. Nakamura, T.: Acquisition of move sequence patterns from game record database using n-gram statistics. In: Game Programming Workshop 1997(1997) (in Japanese)

15. Schaeffer, J., van den Herik, H.J.: Games, Computers, and Artificial Intelligence. Artificial Intelligence 134(1-2), 1–7 (2002)
16. Sinclair, J.: Corpus and text: Basic principles. In: Wynne, M. (ed.) Guide to good practice in developing linguistic corpora (2005)
17. Zipf, G.K.: Human Behavior and the Principle of Least Effort. Addison-Wesley Press, Cambridge (1949)
18. Zobrist, A.: A model of visual organization for the game of go. In: Proceedings AFIPS 34, pp. 103–112 (1969)

A New Proof-Number Calculation Technique for Proof-Number Search

Kazuki Yoshizoe

Quantum Computation and Information Project ERATO-SORST,
Japan Science and Technology Agency, Tokyo, Japan
yoshizoe@acm.org

Abstract. We propose a new simple calculation technique for proof numbers in Proof-Number Search. Search algorithms based on (dis)proof numbers are known to be effective for solving problems such as tsumego, tsume-shogi, and checkers. Usually, the Proof-Number Search expands child nodes with the smallest (dis)proof number because such nodes are expected to be the easiest to (dis)prove the node. However, when many unpromising child nodes exist, (dis)proof numbers are not always a suitable measure for move ordering because unpromising nodes temporarily increase the (dis)proof numbers. For such cases, we propose the use of only some child nodes (instead of all child nodes) for calculating (dis)proof numbers. We call this technique *Dynamic Widening*.

We combined Dynamic Widening with the Depth-first Proof-Number Search (df-pn) algorithm and tested its performance on capturing problems of Go on 19×19 boards. Our results show that the approach is more than 30 times faster than normal df-pn when we generate almost all legal moves (about 300 moves on average). The required time for processing remained approximately four times as long as that of df-pn using heuristic pruning (about 20 moves on average), but the correctness of the search result is guaranteed.

1 Introduction

Search algorithms based on proof numbers are known to be effective for AND/OR tree searches. Depth-first Proof-Number Search (df-pn [6]) is a depth-first version of Proof-Number Search (PNS [1]), which uses both proof numbers and disproof numbers. In fact, df-pn has been useful for solving tsume-shogi [6], tsumego [3], and checkers [7,9] problems.

However, one of its weaknesses is that it is not effective against problems for which the assumption of "a smaller (dis)proof number is promising" does not hold. The initial values of (dis)proof numbers are derived directly from the number of legal moves. Therefore, an intuitive explanation of the proof-number search algorithm is that it is intended to minimize the number of opponent's legal moves.

In tsume-shogi, (dis)proof numbers are extremely useful for move ordering because, after playing a promising move, the number of the opponent's legal

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 135–145, 2008.

moves is typically few. Nevertheless, it is more difficult to solve tsumego because there is usually only a small difference in the number of an opponent's legal moves, whichever move the player chooses. It is difficult even for today's best tsumego solvers [3,12] to solve open-border tsumego problems. Normally, a human player must modify the problems to enclosed problems. If we try to solve open-border tsumego problems, the legal moves are vastly numerous. They appear to be similar no matter which move we choose. For that reason, the initial values of (dis)proof numbers do not reflect the benefits of the moves.

Here, we propose a simple technique, named Dynamic Widening, to address this difficulty. Our method is to use only the few best moves to calculate (dis)proof numbers. Its premise is that, even given numerous legal moves, only the few best moves are important to evaluate the goodness of nodes.

We combined this approach with df-pn+ [6] and measured the performance for Go capturing problems on 19×19 boards. The results showed that it was more than 30 times faster than normal df-pn+ when searching all legal moves. Nevertheless, this approach was not effective when we used heuristic pruning. Searching all legal moves with df-pn+ and Dynamic Widening took 4.1 times longer than using normal df-pn+ with heuristics. However, the advantage of our system is that we can guarantee the correctness of the answers because 3% of the answers from heuristic pruning were wrong.

Section 2 describes related works in this area. The Proof-Number Search basics are described in Sect. 3. Our method is explained in Sect. 4. Section 5 presents the results. Finally, Sect. 6 provides the conclusion of this paper. For information about the game of Go, we refer to `http://www.usgo.org/`.

2 Related Work

GoTools [12] was developed by Thomas Wolf in the early 1990s. It remained the best tsumego solver for more than a decade.

Moreover, df-pn [6] and its variants have been useful for many problems including tsume-shogi solvers, tsumego solver [3], and back-end provers of checkers [9]. A brief explanation of various proof-number search algorithm is given in Sect. 3.

However, neither GoTools nor df-pn variants can solve open-border tsumego problems. Problems must be modified to enclosed problems before being solved by existing solvers. The required size of the enclosed area is becoming larger, but it is still far smaller than a full Go board.

3 Proof-Number Search

In an AND/OR tree, let a *proof* be a win for the first player (corresponding to an OR node) and a *disproof* be a win for the opponent (represented by an AND node). Allis *et al.* [1] introduced proof and disproof numbers as estimates of the difficulty of finding proofs and disproofs in a partially expanded AND/OR tree.

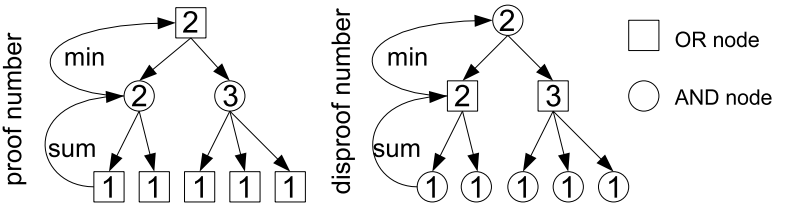


Fig. 1. Calculation of proof/disproof numbers

The proof number of node n, $pn(n)$ is defined as the minimum number of leaf nodes that must be proven to find a proof for n, where the disproof number $dn(n)$ is the minimum number of leaf nodes that must be disproven to find a disproof for n. In addition, let $pn(n) = 0$ and $dn(n) = \infty$ for a proven terminal node n, and $pn(n) = \infty$ and $dn(n) = 0$ for a disproven terminal node. It is also assumed that $pn(n) = dn(n) = 1$ is assigned to any unproven leaf. Let $n_1, \cdots, n_k$ be children of interior node n. Proof and disproof numbers of an OR node n are calculated as follows (shown in Fig. 1).

$$pn(n) = \min_{i=1,\cdots,k} pn(n_i), \qquad dn(n) = \sum_{i=1}^{k} dn(n_i).$$

For an AND node n proof and disproof numbers are

$$pn(n) = \sum_{i=1}^{k} pn(n_i), \qquad dn(n) = \min_{i=1,\cdots,k} dn(n_i).$$

Proof-Number Search (PNS) is a best-first search algorithm that maintains proof and disproof numbers for each node. Actually, PNS finds a leaf node from the root by selecting the child with the smallest proof number at each OR node and the child with the smallest disproof number at each AND node. It then expands that leaf and updates all affected proof and disproof numbers along the path back to the root. This process continues until it finds either a proof or disproof for the root.

Depth-First Proof-Number Search (df-pn) [6] is a depth-first reformulation of PNS that re-expands fewer interior nodes and can run in a space that is limited by the size of the transposition table. Thresholds for proof and disproof numbers are incremented gradually and are used to limit a depth-first search, similar to Recursive Best-First Search [5].

Df-pn+ is used in the best tsume-shogi solver [6]. Df-pn(r) is used for the best tsumego solver [4], and as the back-end prover for solving checkers [8]. Lambda df-pn [13] is a combination of Lambda Search [11] and df-pn. Lambda df-pn was tested for Go capture problems.

4 Proposed Method

4.1 Problem

As explained in Sect. 3, proof-number-based search algorithms expand the nodes with the smallest (dis)proof number. Immediately after the expansion of a node, usually (dis)proof numbers are simply proportional to the number of candidate moves. Negative effects are often observed on move ordering based on (dis)proof numbers when there are numerous (typically unpromising) candidate moves.

For problems such as those of tsume-shogi, proof numbers are quite effective because moves which limit an opponent's legal moves often turn out to be good. However, for problems in games such as Go, the number of an opponent's legal moves does not vary much whichever move the player chooses. Additionally, it is rather difficult to do theoretically safe forward pruning. For example, in capturing problems of Go, heuristic pruning greatly improves the search speed, but the correctness of the result is not guaranteed.

Henceforth, we call a player who tries to prove the tree an *attacker*; the other player is a *defender*.

For OR nodes, if numerous *attacker's* candidate moves exist, the disproof number would be large. In such cases, immediately after expansion of nodes, the difference between disproof numbers would be simply the difference of the number of candidate moves. A technique which delays the generation of unpromising moves is used in existing solvers if it is possible to distinguish unpromising moves from other moves. For tsume-shogi solvers, defensive *dropping moves*[1] are typically delayed. For tsumego, searching for a pass move is often delayed. Therefore, these moves are not included in the calculation of (dis)proof numbers unless other promising moves are disproved.

4.2 Dynamic Widening

In normal proof-number search algorithms, the disproof number of an OR node is calculated as the sum of the disproof numbers of the children (Fig. 2).

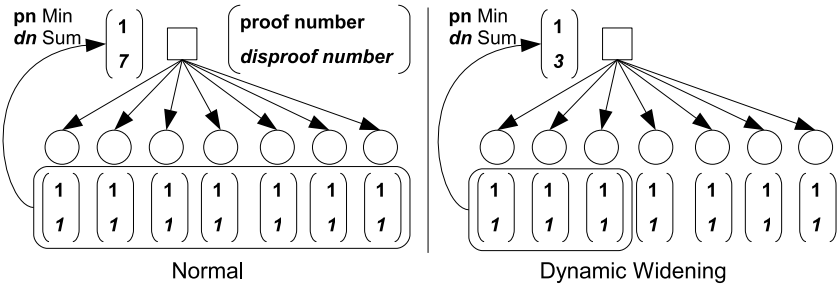


Fig. 2. Dynamic Widening: Immediately after expansion (OR node example)

[1] Reuse of captured opponent pieces.

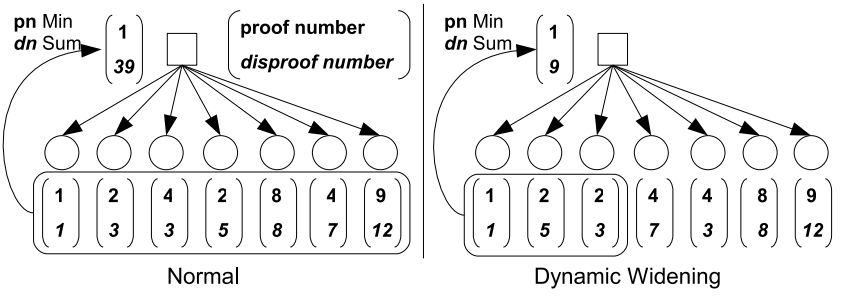


Fig. 3. Dynamic Widening: Search in progress (OR node example)

In this section, our explanation specifically addresses OR nodes, unless otherwise mentioned explicitly. For AND nodes, algorithms are identical if proof numbers and disproof numbers are exchanged.

The technique we propose in this paper is to sum the disproof number of only the best group of the children. As shown in the right tree of the figure, we calculate the disproof number using some child nodes. Figure 2 shows an OR node immediately after the expansion. Initial values of (dis)proof numbers are set to 1.

As the search continues, the (dis)proof numbers will be updated (Fig. 3). Children are sorted according to the proof numbers before calculating the (dis) proof numbers.

Let n be an OR node, and $n_1, \cdots, n_k$ be its children. For OR nodes, n_i are sorted according to their proof numbers in increasing order; only top j children are used to calculate (dis)proof numbers. The (dis)proof number of the OR node n would be

$$pn(n) = \min_{i=1,\cdots,j} pn(n_i), \qquad dn(n) = \sum_{i=1}^{j} dn(n_i).$$

For an AND node, children are sorted in increasing order of disproof numbers. The (dis)proof number will be

$$pn(n) = \sum_{i=1}^{j} pn(n_i), \qquad dn(n) = \min_{i=1,\cdots,j} dn(n_i).$$

If a node is disproven, then the proof number will be set to ∞. After sorting, the disproven node will be passed to the end of the list and another node will become the target of disproof-number calculation. Therefore, the search order will be changed, but the correctness of our technique is maintained.

There might be various ways of defining j. We implemented several methods and measured the performance. Mainly, we compared the performance of the following two methods.

- Set j to a fixed constant ($TOPn$)
- Set j as $1/n$ of all children ($RATEn$)

The search speed might also be improved by assigning small (dis)proof numbers to promising moves, and large (dis)proof numbers to unpromising moves. We combined our methods and this technique (proposed in df-pn+ [6]).

Using this combination, our algorithm will initially use only the promising moves for (dis)proof number calculation. As the search progresses, the algorithm will shift to less-promising moves. Techniques which gradually increase the target scope of the search are called widening [2]. We call our technique Dynamic Widening because the scope for the search changes dynamically during the search.

5 Results and Analysis

5.1 Experimental Conditions

We tested our algorithm for capturing problems of Go on a full 19×19 board. We used the test problem sets distributed by Thomsen [10]. The problem sets do not include problems which result in a Ko. Original problems include double-goal problems (to capture block A or block B). For our purpose, we selected single-goal problems, resulting in 434 problems. For all problems, only one move was determined as the answer. Therefore, we sometimes added answer moves for problems that had multiple answers.

Our algorithm was tested for move generators of two types. The first type generates all legal moves except when the target block is in atari (it generates 300 moves on average). The other type is a heuristic move generator with forward pruning (it generates 20 moves on average). The heuristic move generation uses the idea of *surrounding stones* [11] and consists of (1) the liberties of up to

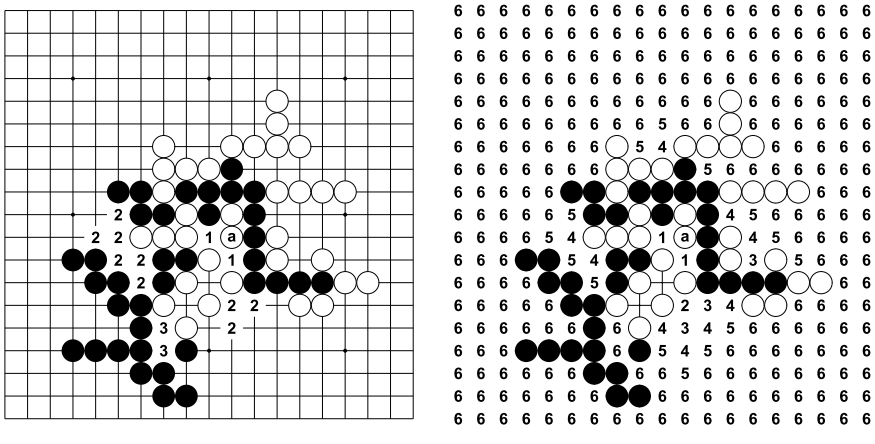


Fig. 4. Example of heuristic move candidates (left) and all legal move candidates (right). The block marked with **"a"** is the capturing target.

5-surrounding stones, (2) liberties of adjacent friend stones, and (3) some other moves including second liberties of target stones.

Figure 4 portrays the difference of the candidate moves. The block marked with an "a" is the target to capture. The intersections marked with a number are used as candidate moves for Black. The number shows the initial value of (dis)proof numbers used in df-pn+. Promising moves have smaller (dis)proof numbers. Let the number in the figure be n. The actual initial (proof number, disproof number) were, for OR nodes, $(2^{(n-1)}, 1)$ for AND nodes, $(1, 2^{(n-1)})$.

We limited the number of node expansions to 200,000 nodes. The nodes actually searched are fewer than the above limit because of re-expansion. The liberty threshold to give up capturing was set to five.

5.2 Answering Ability

The answering abilities of methods are shown in Table 1. The first row shows the move generation method. "All" means that all legal moves are searched (type 1), "heuristic" means that the heuristic move generator is used (type 2). The second row shows the method and parameter used for Dynamic Widening. Also, $TOPn$ means that the best n moves are used in (dis)proof number calculation, and $RATEn$ means that the top $1/n$ moves are used.

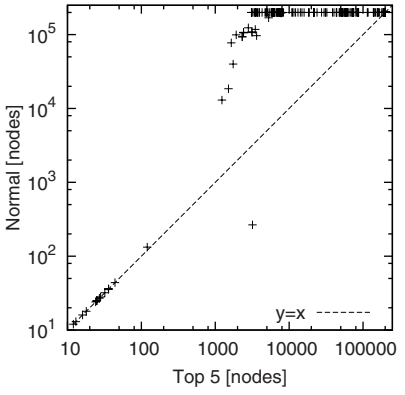
The row named "solved" shows the number of problems solved within 200,000 node expansions; "exceeded" represents the number of problems which were not solved within the threshold. In addition, "miss" signifies the number of problems for which the algorithm returned an answer that turned out to be a wrong move. By searching all legal moves, the returned answers are always right.

The results of Table 1 show that Dynamic Widening is effective if we generate all legal moves. Furthermore, $TOPn$ is improved for smaller n until $n = 5$. However, for $n = 2$, the result is poor. For heuristic pruning, Dynamic Widening has a negative effect. As n becomes smaller, the performance worsens. The heuristic move generator tends to generate fewer candidate moves after promising moves. So, the assumption of "a smaller (dis)proof number means promising" does hold. Here we presume that the worsening performance results from the fact that Dynamic Widening merely neglects this information.

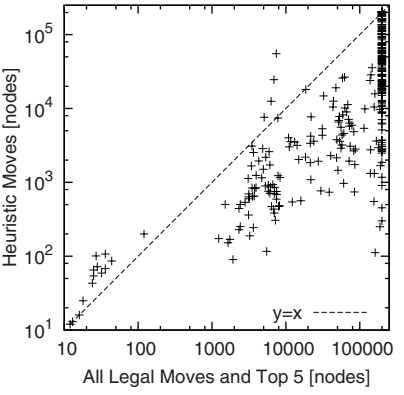
Table 1. Answering ability

cand.	DW method	solved	miss	exceeded
All	none	31	0	403
All	$RATE2$	37	0	397
All	$RATE5$	50	0	384
All	$RATE10$	81	0	353
All	$RATE20$	124	0	310
All	$RATE40$	131	0	303

cand.	DW method	solved	miss	exceeded
All	$TOP20$	108	0	326
All	$TOP10$	138	0	296
All	$TOP5$	139	0	295
All	$TOP2$	8	0	426
heuristic	none	283	9	140
heuristic	$TOP20$	253	9	172
heuristic	$TOP10$	205	8	221
heuristic	$TOP5$	166	9	259



All legal Moves: Normal vs. $TOP5$



Heuristic vs. All Legal ($TOP5$)

Fig. 5. Performance comparison

5.3 Speed Comparison

For ease of comparison of the precise speed performance, we plotted the number of node expansions needed to solve problems. The left chart in Fig. 5 shows a log-log plot of the performance of normal df-pn+ and df-pn+ with $TOP5$ dynamic widening. Each plot shows the number of node expansions necessary to obtain the answer. Most plots are in the upper left area, which shows that, except for a few problems, the $TOP5$ dynamic widening improves the speed performance of df-pn+.

The right chart in Fig. 5 shows a performance comparison between df-pn+ with heuristic pruning, and $TOP5$ dynamic widening when all legal moves are used. As this chart shows, the performance of the $TOP5$ all legal moves df-pn+ is still several times slower than heuristically forward pruned df-pn+.

5.4 Two Sorting Methods

In Table 1, the sorting algorithm for $TOPn$ and $RATEn$ merely considers proof numbers for OR nodes (disproof numbers for AND nodes). Move ordering was done according to the move coordinate when proof numbers were equal. We also tested two slightly different sorting methods. The only difference is in the sorting order when the first comparison is equal.

The first method is to compare the proof number; if the proof number is equal, then select the node with the "greater" disproof numbers. We designate this as the "less -> greater" method. The other is to select the node with a smaller disproof number if the proof number is equal ("less -> less"). (For an explanation of AND nodes, please reverse proof/disproof numbers in this explanation.)

The performance comparison is shown in Fig. 6. A great gap separates the answering ability and the speed performance. At an OR node, a smaller proof number is promising for the *attacker*; a greater disproof number is not promising

sorting method	solved
less -> greater	152
less -> less	128

answering ability

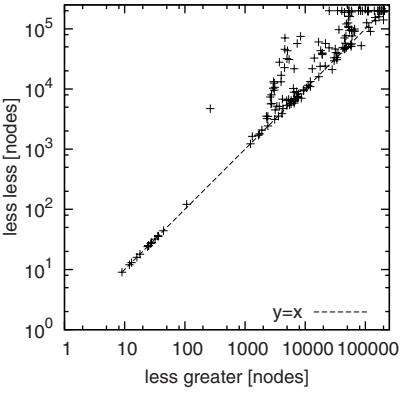


Fig. 6. Answering ability and speed comparison of the sorting method

for the *defender*. This result is explainable simply by the observation that selection of promising moves is the good strategy.

5.5 Overall Analysis

To compare the respective overall performances we took the three methods listed below. We compared the total number of nodes needed to solve the set of problems which was solved using all three methods.

- Default df-pn+ (all legal)
- $TOP5$ dynamic widening df-pn+ (less $->$ greater)
- Forward pruned df-pn+

The result presented in Table 2 shows that $TOP5$ DW makes df-pn+ more than 30 times faster, but it is still slower than df-pn+ using forward pruning. Nevertheless, the advantage of Dynamic Widening is that it can confirm the correctness of the returned answer. In this respect we remark that 3% of the answers returned by the heuristic forward pruning version is wrong.

The children must be sorted before selecting the child to expand. However, we need only sort the top j nodes. Once sorted, the order will not be disturbed so much from the second time and thereafter. Therefore, we expect that sorting is not so time consuming. In this experiment, the time needed for sorting was 0.1–0.2% of the total execution time.

Table 2. Overall performance

method	total nodes	rate1	rate2
df-pn+ (all legal)	2011312	37.4	156
$TOP5$ DW df-pn+	53825	1.0	4.16
Forward Pruned df-pn+	12934	0.240	1.0

In conclusion, Dynamic Widening is effective for capturing problems of Go if all legal moves are considered. It can be implemented easily with only a small overhead of execution time. It was not effective when combined with heuristic forward pruning, but will be useful for problems with large branching factors such as those of Go. Dynamic Widening is particularly effective in addressing problems for which it is necessary to guarantee correctness or those for which heuristic pruning is difficult.

6 Conclusions and Future Work

We proposed a simple technique for Proof-Number Search: Dynamic Widening. Although it is tested for df-pn only, the method is expected to be effective for other proof-number search variants such as PNS [1].

From the experimental results we may conclude that this method is effective for capturing problems of Go if all legal moves are searched. We expect that this method is suitable for other problems with large branching factors and that theoretically safe forward pruning is difficult. Particularly, we expect that the new method is useful to solve open-border tsumego problems.

In the future, more experiments should be performed for problems related to other games and domains. Additionally, we plan to improve the speed and combine this method with other algorithms to test the capabilities of Dynamic Widening further.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-number search. Artificial Intelligence 66(1), 91–124 (1994)
2. Cazenave, T.: Generalized widening. In: Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI 2004), pp. 156–160 (2004)
3. Kishimoto, A.: Correct and Efficient Search Algorithms in the Presence of Repetitions. PhD thesis, University of Alberta (March 2005)
4. Kishimoto, A., Müller, M.: Search versus knowledge for solving life and death problems in Go. In: Proc. of Twentieth National Conference on Artificial Intelligence (AAAI 2005), pp. 1374–1379. AAAI Press, Menlo Park (2005)
5. Korf, R.E.: Linear-space best-first search. Artificial Intelligence 62(1), 41–78 (1993)
6. Nagai, A.: Df-pn Algorithm for Searching AND/OR Trees and Its Applications. PhD thesis, University of Tokyo, Tokyo (2002)
7. Schaeffer, J.: Game over: Black to play and draw in checkers. ICGA Journal 30(4), 187–197 (2007)
8. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Solving checkers. In: Proc. of 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 292–297 (2005)
9. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science 317(5844), 1518–1522 (2007)
10. Thomsen, T.: Madlab website, http://www.t-t.dk/madlab/problems/index.html

11. Thomsen, T.: Lambda-search in game trees - with application to Go. ICGA Journal 23(4), 203–217 (2000)
12. Wolf, T.: Forward pruning and other heuristic search techniques in tsume go. Special issue of Information Sciences 122(1), 59–76 (2000)
13. Yoshizoe, K., Kishimoto, A., Müller, M.: Lambda depth-first proof number search and its application to go. In: Proc. of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2404–2409 (2007)

About the Completeness of Depth-First Proof-Number Search

Akihiro Kishimoto[1] and Martin Müller[2]

[1] Department of Media Architecture, Future University-Hakodate 116-2, Kamedanakano, Hakodate, Hokkaido, 041-8655, Japan
kishi@fun.ac.jp
[2] Department of Computing Science, University of Alberta, Edmonton, AB, Canada, T6G 2E8
mmueller@cs.ualberta.ca

Abstract. Depth-first proof-number (df-pn) search is a powerful member of the family of algorithms based on proof and disproof numbers. While df-pn has succeeded in practice, its theoretical properties remain poorly understood. This paper resolves the question of completeness of df-pn: its ability to solve any finite boolean-valued game tree search problem in principle, given unlimited amounts of time and memory. The main results are that df-pn is complete on finite directed acyclic graphs (DAG) but incomplete on finite directed cyclic graphs (DCG).

1 Introduction

AND/OR tree search is a standard technique for determining the winner of a two-player game. Research in tree search methods has led to remarkable progress over the last 15 years, especially on algorithms based on proof and disproof numbers [1]. Nagai's depth-first proof-number search (df-pn) is a particularly attractive method [6] which has been applied successfully in games such as shogi [6], Go [5], and checkers [10].

One important open question concerns the *completeness* of df-pn. An algorithm $\mathcal{A}$ to solve problems in domain $\mathcal{D}$ is called *complete* if $\mathcal{A}$ can eventually solve any problem in $\mathcal{D}$, and *incomplete* otherwise. Resolving the completeness of df-pn is relevant both for theory and practical applications. Completeness implies that there are no fundamental obstacles for solving increasingly hard problems. This paper proves two main results.

1. Df-pn is complete on finite directed acyclic graphs (DAGs), given unlimited amounts of time and memory.
2. Df-pn is incomplete on finite directed cyclic graphs (DCGs). There exist problems that df-pn cannot solve. A concrete counterexample is given.

The structure of this paper is as follows. Sections 2 and 3 review terminology for AND/OR trees, proof number search, and df-pn. Sections 4 and 5 prove the completeness of df-pn on DAGs and its incompleteness on DCGs.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 146–156, 2008.

2 AND/OR Trees and Graphs

An AND/OR tree contains two types of nodes that assume dual roles, namely *OR* and *AND* nodes. In the standard model, all children of an OR node are AND nodes, and vice versa. The *root* is the only node that has no parent. It is assumed to be an OR node.

Each node can assume three kinds of values: **true**, **false**, and **unknown**. In a solved AND/OR tree, the value of the root is either **true** or **false**. A *terminal* node has no children. Its value must also be either **true** or **false**. A node with at least one child is called an *interior* node. A *leaf* node has not yet been expanded. It could be either terminal or interior. The phrase "a node is x" will be used as a short form for "a node has value x".

An interior OR node is **true** if at least one of its children is **true**, it is **false** if all its children are **false**, and **unknown** otherwise. Likewise, an interior AND node is **false** if at least one of its children is **false**, **true** if all children are **true**, and **unknown** otherwise. A **true** node is also called a *proven* node, and a **false** node is called *disproven*. A *(dis)proof* demonstrates that a node is (dis)proven. A *proof tree* T provides a proof for a node n. It contains (1) n, (2) at least one child of each interior OR node of T, and (3) all children of interior AND nodes of T. All terminal nodes of T must be proven. *Disproof trees* are defined analogously.

In contrast to a tree, a *directed acyclic graph (DAG)* may contain more than one path between two nodes. The value of each node is well-defined by applying the same rules as for trees in a bottom-up manner. The other concepts defined above for trees carry over to DAGs as well.

Cycles in a *directed cyclic graph (DCG)* potentially cause recursive dependencies of a node value on itself. Therefore, values are usually defined in a game-specific way that takes into account the *history*, the path taken to reach a node. A node in a DCG implicitly represents the different paths taken to reach that node. The game outcome may differ depending on which path was taken. This phenomenon is known as the Graph History Interaction (GHI) problem [2,8].

3 Proof-Number Search and Df-pn

3.1 Proof-Number Search

Allis' proof-number search (PNS) is a best-first search algorithm based on proof and disproof numbers [1]. These numbers estimate how easy it is to prove or disprove a node by further expanding an AND/OR tree. The (dis)proof number $\mathbf{pn}(n)$ ($\mathbf{dn}(n)$) of node n is the minimum number of leaf nodes that must be (dis)proven to (dis)prove n.

Let $n_1, n_2, \cdots, n_k$ be children of node n. Since only one proven child suffices to prove an OR node, while all children must be proven to prove an AND node (and vice versa for disproof), $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ are calculated as follows:

1. For a proven node n, $\mathbf{pn}(n) = 0$ and $\mathbf{dn}(n) = \infty$.
2. For a disproven node n, $\mathbf{pn}(n) = \infty$ and $\mathbf{dn}(n) = 0$.

3. For an unknown leaf node n, $\mathbf{pn}(n) = \mathbf{dn}(n) = 1$.
4. For an interior OR node n, $\mathbf{pn}(n) = \min_{i=1,2,\cdots,k} \mathbf{pn}(n_i)$, $\mathbf{dn}(n) = \sum_{i=1}^{k} \mathbf{dn}(n_i)$.
5. For an interior AND node n, $\mathbf{pn}(n) = \sum_{i=1}^{k} \mathbf{pn}(n_i)$, $\mathbf{dn}(n) = \min_{i=1,2,\cdots,k} \mathbf{dn}(n_i)$.

PNS expands a *most promising* leaf node by selecting a child with smallest (dis)proof number at OR (AND) nodes, starting from the root, then updates proof and disproof numbers along the path back to the root. The process continues until either a proof or disproof of the root is found, or resources are exhausted.

3.2 Depth-First Proof-Number Search

The depth-first proof-number (df-pn) search algorithm [6] is a depth-first variant of PNS which expands fewer interior nodes and requires less memory than PNS. Df-pn utilizes thresholds for proof and disproof numbers. If either $\mathbf{pn}(n)$ or $\mathbf{dn}(n)$ exceeds its respective threshold, a most-promising node cannot exist in the subtree below n. See [9] for a clear exposition of df-pn.

Figure 1, adapted from [6], presents pseudo-code of the df-pn algorithm[1]. The code is written in negamax fashion to avoid two dual cases. For node n, $\phi(n)$ and $\delta(n)$ are defined as follows:

$$\phi(n) = \begin{cases} \mathbf{pn}(n) \ (n \text{ is an OR node}) \\ \mathbf{dn}(n) \ (n \text{ is an AND node}), \end{cases}$$

$$\delta(n) = \begin{cases} \mathbf{dn}(n) \ (n \text{ is an OR node}) \\ \mathbf{pn}(n) \ (n \text{ is an AND node}). \end{cases}$$

The computation of $\phi(n)$ and $\delta(n)$ for an interior node n simply becomes:

$$\phi(n) = \min_{i=1,2,\cdots,k} \delta(n_i), \quad \delta(n) = \sum_{i=1}^{k} \phi(n_i).$$

Nagai proved that df-pn is equivalent to PNS in the sense that both algorithms always expand a most promising node [6]. However, the proof requires the search space to be a tree. If the search space is not a tree, no theoretical property such as completeness has been shown. PNS is complete on a finite search space, because it keeps traversing nodes until it either expands a leaf or detects a cycle. However, df-pn may exceed the thresholds without reaching any leaf on DAGs or DCGs. The next section describes this problem.

[1] The original code in [6] saves the proof and disproof number thresholds of a node in the transposition table before expanding that node. However, this step can be omitted for efficiency [7].

```
// Set up for the root node
// Note that the root is an OR node
int Df-pn(node r) {
  r.φ = ∞;  r.δ = ∞;
  MID(r);
  if (r.δ = ∞)
    return true;
  else
    return false;
}
// Iterative deepening at each node
void MID(node n) {
  TTlookup(n,φ,δ);
  if (n.φ ≤ φ  ||  n.δ ≤ δ)  {
    // Exceed thresholds
    n.φ = φ; n.δ = δ;
    return;
  }
  // Terminal node
  if (IsTerminal(n)) {
    Evaluate(n);
    // Store (dis)proven node
    TTstore(n,n.φ,n.δ);
    return;
  }
  GenerateMoves(n);
  // Iterative deepening
  while (n.φ > ΔMin(n)  &&
          n.δ > ΦSum(n))  {
    n_c = SelectChild(n,φ_c,δ_c,δ_2);
    // Update thresholds
    n_c.φ = n.δ + φ_c - ΦSum(n);
    n_c.δ = min(n.φ,δ_2 + 1);
(+) MID(n_c);
  }
  // Store search results
  n.φ = ΔMin(n); n.δ = ΦSum(n);
  TTstore(n,n.φ,n.δ);
}
```

```
// Select the most promising child
node SelectChild(node n, int &φ_c,
                 int &δ_c, int &δ_2)  {
  node n_best;
  δ_c = φ_c = ∞;
  for (each child n_child) {
    TTlookup(n_child,φ,δ);
    // Store the smallest and second
    // smallest δ in δ_c and δ_2
    if (δ < δ_c) {
      n_best = n_child;
      δ_2 = δ_c; φ_c = φ; δ_c = δ;
    }
    else if (δ < δ_2)
      δ_2 = δ;
    if (φ = ∞)
      return n_best;
  }
  return n_best;
}
// Compute the smallest δ of
// n's children
int ΔMin(node n) {
  int min = ∞;
  for (each child n_child) {
    TTlookup(n_child,φ,δ);
    min = min(min,δ);
  }
  return min;
}
// Compute sum of φ of n's children
int ΦSum(node n) {
  int sum = 0;
  for (each child n_child) {
    TTlookup(n_child,φ,δ);
    sum = sum + φ;
  }
  return sum;
}
```

Fig. 1. Pseudo-code of the df-pn algorithm

4 Depth-First Proof-Number Search in DAGs

4.1 Problem Description

Although completeness on finite DAGs is a seemingly natural property of game tree search algorithms, it is not a trivial question for the case of df-pn for the following reason. If df-pn expands a node n via path p, it backs up proof and disproof numbers only along p. Nodes along other paths to n are not updated

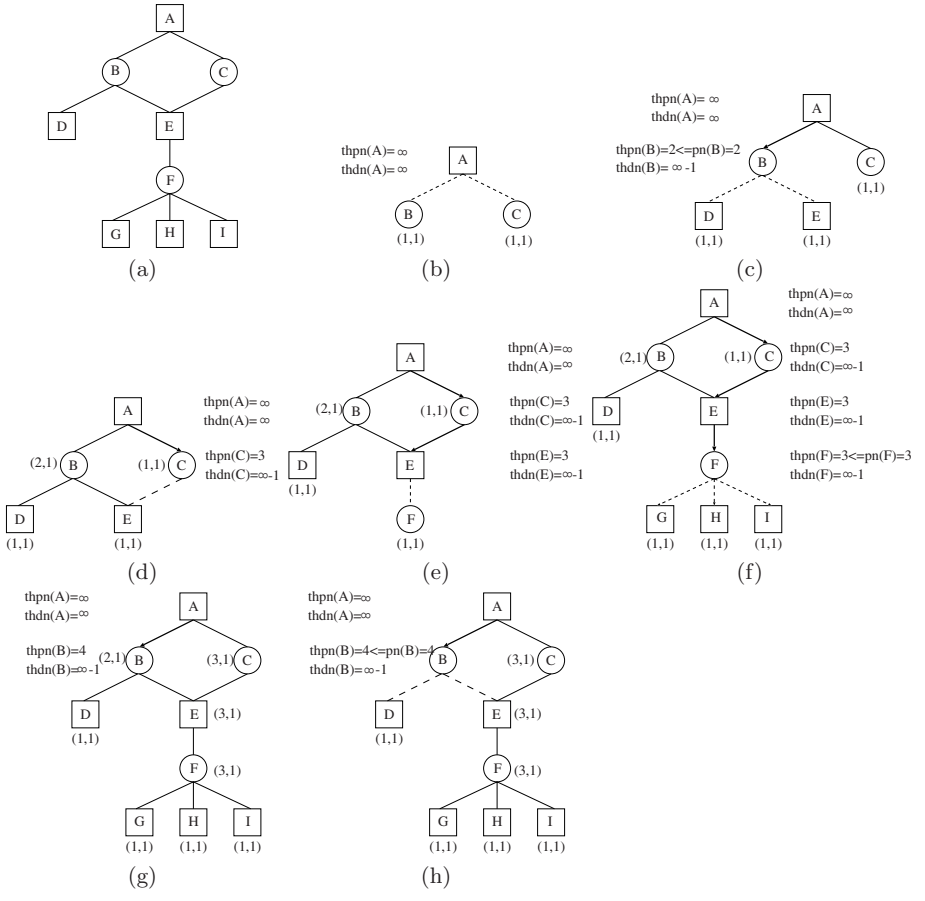


Fig. 2. An example in which df-pn does not expand a leaf node

to become consistent with the changes in $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$. This is a pragmatic choice: there may be exponentially many such paths. While path length is typically logarithmic in game tree size S, the number of nodes on all paths to n may be linear in S.

Because of inconsistent node values, a call of MID (see Fig. 1) in df-pn may return without expanding a leaf node.

Figure 2 shows an example, with proof and disproof numbers written as pairs (pn, dn). The DAG that df-pn explores is shown in Fig. 2(a). $\mathbf{th_{pn}}(n)$ and $\mathbf{th_{dn}}(n)$ are the current thresholds for proof and disproof numbers. A dotted line indicates that df-pn recomputes $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$ by checking proof and disproof numbers of n's children. If n is a leaf node, a dotted line indicates that df-pn expands n to recompute $\mathbf{pn}(n)$ and $\mathbf{dn}(n)$. Ties are broken in favor of the leftmost node.

In the example, the subtree below E is first explored through C as shown in Figs. 2(b)-(f). Next, df-pn selects B with $\mathbf{th_{pn}}(B) = 4$ (see Fig. 2(g)). When $\mathbf{pn}(B)$ is computed, B's child E already has a proof number of 3. Therefore $\mathbf{pn}(B) = 4 \geq \mathbf{th_{pn}}(B) = 4$ (Fig. 2(h)), and MID returns immediately without reaching a leaf node.

4.2 Completeness of Depth-First Proof-Number Search

A search is complete if it is eventually able to find a (dis)proof of the root. This is guaranteed if in the worst case the whole search space can be explored. Whenever a leaf node is expanded, the unexplored part of the search space shrinks. Therefore, the only potential problem is when df-pn exceeds a threshold at an interior node and does not expand a leaf. This section shows that in this case, df-pn makes progress with updating the proof and disproof numbers. They become more consistent, which eventually leads to the expansion of a leaf node. For brevity, DAG always stands for a finite DAG in this section.

Definition 1. *(Consistent and inconsistent nodes) Interior node n of DAG G with children $n_1, n_2, \cdots, n_k$ is called consistent if $\phi(n) = \min_{i=1,2,\cdots,k} \delta(n_i)$ and $\delta(n) = \sum_{i=1}^{k} \phi(n_i)$, otherwise n is called inconsistent.*

Definition 2. *(Level) The level $l(n)$ of node n in DAG G is the maximum length of any path from the root to n in G.*

Definition 3. *(Consistent Graph) A DAG G is called consistent from level l, or short l-consistent, if any interior node n with level $l(n) \geq l$ in G is consistent. G is called consistent if it is 0-consistent.*

Definition 4. *(Inconsistency tuple) Let $|N_l|$ be the number of inconsistent interior nodes at level l in DAG G, and l_{max} be the maximum level of all interior nodes. The inconsistency tuple $IT(G)$ is defined as $(|N_{l_{max}}|, |N_{l_{max}-1}|, \cdots, |N_0|)$. We will write $IT(G) = 0$ short for $IT(G) = (0, \cdots, 0)$.*

Note that G is consistent if and only if $IT(G) = 0$.

Definition 5. *($<_{lex}$) Let $i_0, i_1, \cdots, i_l$ and $j_0, j_1, \cdots, j_l$ be nonnegative integers. The lexicographical order $<_{lex}$ is defined as follows:*

$$(i_0, i_1, \cdots, i_l) <_{lex} (j_0, j_1, \cdots, j_l) \Longleftrightarrow \exists k \text{ s.t. } i_0 = j_0, i_1 = j_1, \cdots, i_{k-1} = j_{k-1} \text{ and } i_k < j_k.$$

The following lemma guarantees that each run of MID that does not expand a leaf node lexicographically decreases $IT(G)$. Such a run changes at least one inconsistent node to be consistent.

Lemma 1. *Assume that unproven DAG G is inconsistent and that df-pn is about to call MID(n_c) (see (+) in Fig. 1) for a child n_c of node n. Let T and U be the values of $IT(G)$ before and after the run of MID(n_c) respectively. If MID(n_c) expands no leaf node, then $U <_{lex} T$.*

Proof. Let $\mathbf{th}_\phi(n)$ and $\mathbf{th}_\delta(n)$ be the thresholds of ϕ and δ at node n. The run of MID(n_c) encounters at least one interior node m at which df-pn satisfies the termination condition $\mathbf{th}_\phi(m) \leq \Delta Min(m)$ or $\mathbf{th}_\delta(m) \leq \Phi Sum(m)$. Let o be an interior node of which the level $l(o)$ is largest among all such m. Let T and U be $(t_{l_{max}}, t_{l_{max}-1}, \cdots, t_0)$ and $(u_{l_{max}}, u_{l_{max}-1}, \cdots, u_0)$ respectively. Because G is a DAG, $u_k = t_k$ holds for $l(o) < k \leq l_{max}$. Moreover, since computing $\Delta Min(o)$ and $\Phi Sum(o)$ in Fig. 1 fixes the inconsistency at o, $u_{l(o)} < t_{l(o)}$, which implies $U <_{lex} T$. □

The following two lemmas prove properties of DAGs that are consistent from level l.

Lemma 2. *Let $l = l(n)$, and $\mathbf{th}_\phi(n)$ and $\mathbf{th}_\delta(n)$ be the thresholds of ϕ and δ at node n. Assume that df-pn is at interior node n with $\mathbf{th}_\phi(n) > \phi(n)$ and $\mathbf{th}_\delta(n) > \delta(n)$ in unproven l-consistent DAG G. Then, df-pn selects a child n_c of n with $\mathbf{th}_\phi(n_c) > \phi(n_c)$ and $\mathbf{th}_\delta(n_c) > \delta(n_c)$.*

Proof. Because G is l-consistent, $\phi(n) = \min_{i=1,2,\cdots,k} \delta(n_i)$ and $\delta(n) = \sum_{i=1}^{k} \phi(n_i)$. When df-pn recomputes $\delta(n)$ and $\phi(n)$ by retrieving proof and disproof numbers of n's children from the transposition table, $\delta(n)$ and $\phi(n)$ remain unchanged, and df-pn still does not satisfy the termination condition. Let n_c be a node of which δ is the smallest and n_{sec} be one of which δ is the second smallest. Note that ϕ of n corresponds to δ of n_c and n_{sec} because of the duality of proof and disproof numbers. Df-pn selects n_c with the following thresholds:

$$\mathbf{th}_\phi(n_c) = \mathbf{th}_\delta(n) + \phi(n_c) - \sum_{i=1}^{k} \phi(n_i), \ \mathbf{th}_\delta(n_c) = \min(\mathbf{th}_\phi(n), \delta(n_{sec}) + 1).$$

Because $\mathbf{th}_\delta(n) > \delta(n)$ and $\delta(n) = \sum_{i=1}^{k} \phi(n_i)$, $\mathbf{th}_\phi(n_c) > \phi(n_c)$ holds. Also, because $\delta(n_{sec})+1 > \delta(n_c)$ and $\mathbf{th}_\phi(n) > \phi(n) = \delta(n_c)$, $\mathbf{th}_\delta(n_c) > \delta(n_c)$ holds. □

Lemma 3. *Assume that df-pn is at the call MID(n) and $l = l(n)$ in unproven l-consistent DAG G. If $\mathbf{th}_\phi(n) > \phi(n)$ and $\mathbf{th}_\delta(n) > \delta(n)$, then df-pn finds and expands a leaf.*

Proof. Because G is l-consistent, Lemma 2 can be recursively applied to the sequence of interior nodes that df-pn selects starting from n. Because G is finite, this sequence must reach a leaf n_{leaf}. By lemma 2, $\mathbf{th}_\phi(n_{leaf}) > \phi(n_{leaf})$ and $\mathbf{th}_\delta(n_{leaf}) > \delta(n_{leaf})$. Hence, df-pn expands n_{leaf}. □

Theorem 1. *The df-pn algorithm is complete on any finite DAG.*

Proof. We show that there exists no node that causes df-pn to loop forever in the while loop of MID in Fig. 1.

Assume that such a node n exists in the current DAG G. The only case that must be considered is when df-pn expands no leaf node below n and always satisfies $\mathbf{th}_\phi(n) > \Delta Min(n)$ and $\mathbf{th}_\delta(n) > \Phi Sum(n)$. If df-pn expands a leaf reached from n, the number of unexpanded nodes is reduced and the while loop will eventually be exited.

By Lemma 3, if no leaf is expanded, G can not be $l(n)$-consistent. Assume that df-pn is at n and let T and U be the values of $IT(G)$ before and after df-pn calls MID on the line marked by (+) in Fig. 1. By applying Lemma 1, $U <_{lex} T$ holds. This indicates $IT(G) = 0$ after a finite number of MID calls. Because G eventually becomes consistent, it also becomes $l(n)$-consistent. By Lemma 3 df-pn expands a leaf node below n, which is a contradiction.

Because $\mathbf{th}_\phi(r) = \infty$ and $\mathbf{th}_\delta(r) = \infty$ at the root r and either $\phi(r) \geq \mathbf{th}_\phi(r)$ or $\delta(r) \geq \mathbf{th}_\delta(r)$ holds after a finite number of steps, df-pn eventually solves r. □

5 Depth-First Proof-Number Search in DCGs

5.1 Incompleteness of df-pn

Df-pn can only be used on DCGs when the Graph History Interaction (GHI) problem [2,8] is handled. Even though a correct and efficient solution to GHI exists [4], df-pn is not a complete algorithm.

Theorem 2. *Df-pn is incomplete on DCGs.*

Proof. We prove that in the problem shown in Fig. 3(a), which is simplified from an example in [3], df-pn loops forever among nodes A-O and never expands P. Figures 3(b)-(i) show the crucial steps in running df-pn. Let k be the number of visits to node C in Fig. 3(a), l be a positive integer, and $\mathbf{pn}_k(n)$, $\mathbf{dn}_k(n)$ be proof and disproof numbers of node n at the kth visit to C. As before, ties are broken left to right.

Claim: the following equations hold:

$$\mathbf{pn}_k(X) = 1 \quad \textit{for all } k \textit{ and all } X \in \{F, G, H, I, J, K, L, M, N, O, P\}$$

$$\mathbf{dn}_k(X) = 1 \quad \textit{for all } k \textit{ and all } X \in \{I, K, N, P\}$$

$$\mathbf{dn}_k(F) = \begin{cases} 1, & (\textit{for } k = 1) \\ k, & (\textit{for } k = 2l) \\ k-1, & (\textit{for } k = 2l+1) \end{cases}$$

$$\mathbf{dn}_k(G) = \begin{cases} k, & (\textit{for } k = 2l-1) \\ k-1, & (\textit{for } k = 2l) \end{cases}$$

$$\mathbf{dn}_k(H) = \mathbf{dn}_k(L) = \begin{cases} 1, & (\textit{for } k = 1) \\ k-1, & (\textit{for } k = 2l) \\ k-2, & (\textit{for } k = 2l+1) \end{cases}$$

$$\mathbf{dn}_k(J) = \mathbf{dn}_k(O) = \begin{cases} 1, & (\textit{for } k = 1, 2) \\ k-1, & (\textit{for } k = 2l+1) \\ k-2, & (\textit{for } k = 2l+2) \end{cases}$$

$$\mathbf{dn}_k(M) = \begin{cases} 1, & (\textit{for } k = 1, 2, 3) \\ k-2, & (\textit{for } k = 2l+2) \\ k-3, & (\textit{for } k = 2l+3). \end{cases}$$

Proof: by induction on k.

- *(Case $k \leq 5$) Figures 3(b)-(i) give a trace of df-pn on the graph in Fig. 3(a). In this figure, (c), (e), (g), and (i) correspond to $k = 1, 2, 3$, and 4 respectively. P is not explored for $k \leq 4$. By inspection, the claim holds for each node in these figures. The analogous proof for the case of $k = 5$ is omitted here for lack of space.*
- *Assume that the claim holds for k. It is proven for $k+1$ by tracing the search graph.*
  - *(Case $k = 2l + 2$) At the kth visit to C, $\mathbf{pn}_k(F) = \mathbf{pn}_k(G) = 1$ and $\mathbf{dn}_k(F) = k > \mathbf{dn}_k(G) = k-1$. G is, therefore, chosen with $\mathbf{th}_{\mathbf{pn}}(G) = 1$ and $\mathbf{th}_{\mathbf{dn}}(G) = k+1$. At G, $\mathbf{th}_{\mathbf{pn}}(G) = 2 \geq \min(\mathbf{pn}_k(J), \mathbf{pn}_k(K)) = 1$ and $\mathbf{th}_{\mathbf{dn}}(G) = k+1 \geq \mathbf{dn}_k(J) + \mathbf{dn}_k(K) = k-2+1 = k-1$, G is expanded. Since $\mathbf{pn}_k(J) = \mathbf{pn}_k(K) = 1$, J is chosen with $\mathbf{th}_{\mathbf{pn}}(J) = 2$ and $\mathbf{th}_{\mathbf{dn}}(J) = k$. At J, O is explored since $\mathbf{th}_{\mathbf{pn}}(J) = 2 \geq \min(\mathbf{pn}_k(O)) = 1$ and $\mathbf{th}_{\mathbf{dn}}(J) = k \geq \mathbf{dn}_k(O) = k-2$. Therefore, O is selected to expand with $\mathbf{th}_{\mathbf{pn}}(O) = 2$ and $\mathbf{th}_{\mathbf{dn}}(O) = k$. However, the termination condition is satisfied at O, because $\mathbf{dn}_k(H) + \mathbf{dn}_k(P) = k-1+1 = k \geq \mathbf{th}_{\mathbf{dn}}(O) = k$. Thus, P is not explored and proof and disproof numbers are backed up to C as follows:*

$$\begin{aligned} \mathbf{pn}_{k+1}(O) &= \min(\mathbf{pn}_k(H), \mathbf{pn}_k(P)) = 1 \\ \mathbf{dn}_{k+1}(O) &= \mathbf{dn}_k(H) + \mathbf{dn}_k(P) = k \\ \mathbf{pn}_{k+1}(J) &= \mathbf{pn}_{k+1}(O) = 1 \\ \mathbf{dn}_{k+1}(J) &= \mathbf{dn}_{k+1}(O) = k \\ \mathbf{pn}_{k+1}(G) &= \min(\mathbf{pn}_{k+1}(J), \mathbf{pn}_k(K)) = 1 \\ \mathbf{dn}_{k+1}(G) &= \mathbf{dn}_{k+1}(J) + \mathbf{dn}_k(K) = k+1. \end{aligned}$$

    For the remaining nodes F, H, I, K, L, M, and N, proof and disproof numbers remain the same. Thus, the claim is proven for $k+1$.
  - *(Case $k = 2l + 3$) This case is proven with an analogous discussion. At the kth visit to C, $\mathbf{pn}_k(F) = \mathbf{pn}_k(G) = 1$ and $\mathbf{dn}_k(F) = k-1 < \mathbf{dn}_k(G) = k$. F is chosen with $\mathbf{th}_{\mathbf{pn}}(F) = 2$ and $\mathbf{th}_{\mathbf{dn}}(F) = k+1$. $C \to F \to H \to L \to M$ is explored. The termination condition at M holds, since $\mathbf{dn}_k(O) = k-1 \geq \mathbf{th}_{\mathbf{dn}}(M) = k-1$. Proof and disproof numbers are backed up to C and change as in the equations above.*

P is never explored and df-pn loops forever. □

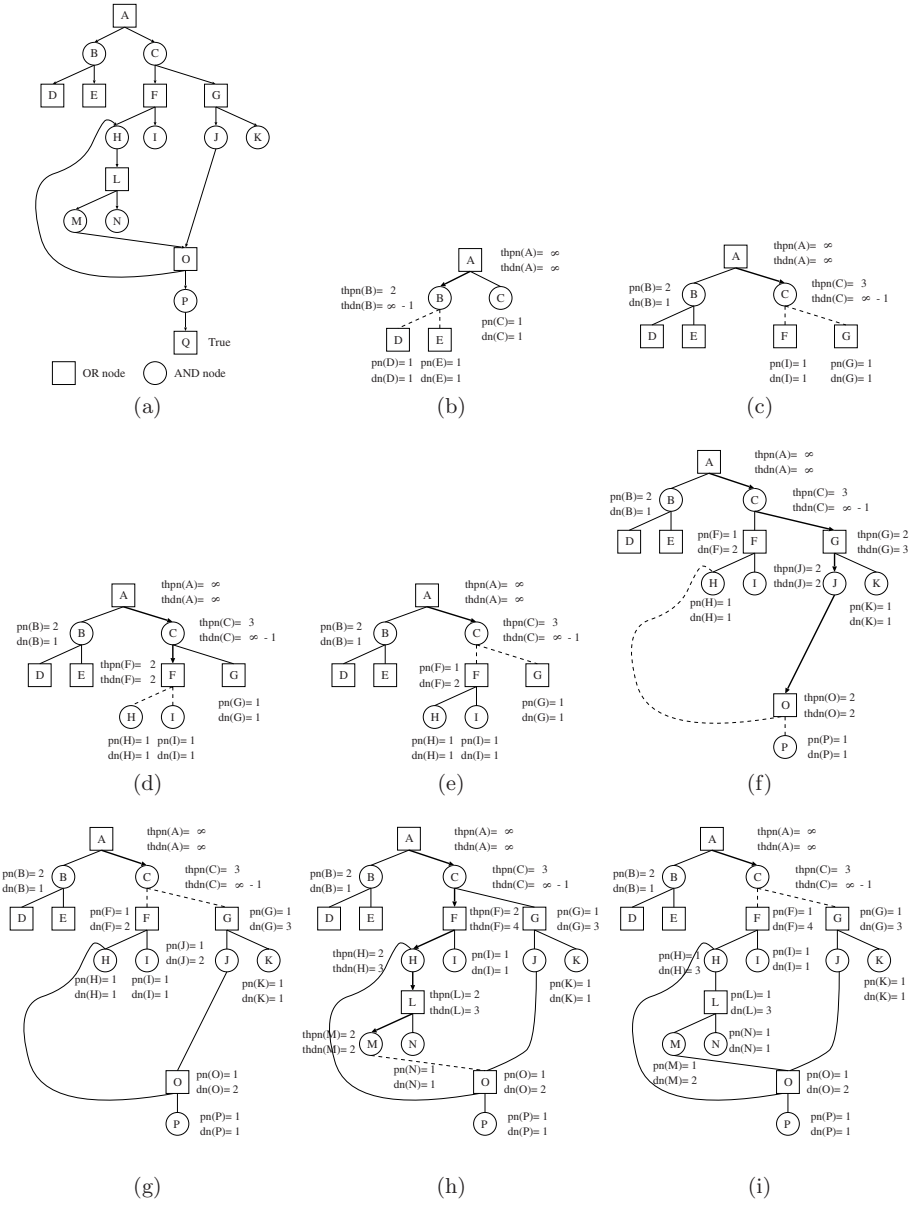


Fig. 3. An example in which df-pn loops forever

5.2 Df-pn(r)

Df-pn(r) [3] is an improved version of df-pn that avoids the looping behavior in cases such as Fig. 3. It modifies the computation of proof and disproof numbers by omitting so-called *old* children such as H when computing $\mathbf{dn}(O)$. Because

only old nodes may lead to repetitions, this type of modification seems to be a reasonable attempt towards designing a complete algorithm. While infinite loops have never been observed when using df-pn(r), even on very complex problems in Go and checkers [3], the question of completeness of this algorithm on DCGs remains unresolved.

6 Conclusion and Future Work

This paper established the completeness properties of depth-first proof-number search. The result that df-pn can solve any finite problem on DAGs is encouraging. However, since df-pn is shown to be incomplete on finite DCGs, the search for complete versions of the algorithm becomes urgent. Df-pn(r) seems to be a strong candidate.

Acknowledgments. This research was financially supported by NSERC, the Natural Sciences and Engineering Research Council of Canada and iCORE, the Alberta Informatics Circle of Research Excellence.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-number search. Artificial Intelligence 66(1), 91–124 (1994)
2. Campbell, M.: The graph-history interaction: On ignoring position history. In: 1985 Association for Computing Machinery Annual Conference, pp. 278–280 (1985)
3. Kishimoto, A.: Correct and Efficient Search Algorithms in the Presence of Repetitions. PhD thesis, Department of Computing Science, University of Alberta (2005)
4. Kishimoto, A., Müller, M.: A general solution to the graph history interaction problem. In: 19th National Conference on Artificial Intelligence (AAAI 2004), pp. 644–649. AAAI Press, Menlo Park (2004)
5. Kishimoto, A., Müller, M.: Search versus knowledge for solving life and death problems in Go. In: Twentieth National Conference on Artificial Intelligence (AAAI 2005), pp. 1374–1379. AAAI Press, Menlo Park (2005)
6. Nagai, A.: Df-pn Algorithm for Searching AND/OR Trees and Its Applications. PhD thesis, Department of Information Science, University of Tokyo (2002)
7. Nagai, A.: Private communication (2005)
8. Palay, A.J.: Searching with Probabilities. PhD thesis, Carnegie Mellon University (1983); Also published by Pitman (1985)
9. Pawlewicz, J., Lew, L.: Improving depth-first PN-search: $1 + \epsilon$ trick. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 160–171. Springer, Heidelberg (2007)
10. Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science 317(5844), 1518–1522 (2007)

Weak Proof-Number Search

Toru Ueda, Tsuyoshi Hashimoto, Junichi Hashimoto, and Hiroyuki Iida

School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{s0610013,t-hashi,j-hashi,iida}@jaist.ac.jp

Abstract. The paper concerns an AND/OR-tree search algorithm to solve hard problems. Proof-number search is a well-known powerful search algorithm for that purpose. Its depth-first variants such as PN*, PDS, and df-pn work very well, in particular in the domain of shogi mating problems. However, there are still possible drawbacks. The most prevailing one is the double-counting problem. To handle this problem the paper proposes a new search idea using proof number and branching factor as search estimators. We call the new method *Weak Proof-Number Search*. The experiments performed in the domain of shogi and Othello show that the proposed search algorithm is potentially more powerful than the original proof-number search or its depth-first variants.

1 Introduction

In 1994, Allis developed the proof-number search (PN-search) algorithm [2] for finding the game-theoretical value in game trees. PN-search is a best-first search, in which the cost function used in deciding which node to expand next is given by the minimum number of nodes that have to be expanded to prove the goal. As such it is a successor of conspiracy-number search [11,19]. PN-search is appropriate in cases where the goal is a well-defined predicate, such as proving a game to be a first-player win.

PN-search can be a powerful game solver in various simple domains such as connect-four and qubic. Its large disadvantage is that, as a genuine best-first algorithm, it uses a large amount of memory, since the complete search tree has to be kept in memory. To handle the memory disadvantage of PN-search, PN* was proposed [22]. It is a search algorithm for AND/OR tree search, which is a depth-first alternative for PN-search. The idea was derived from Korf's RBFS algorithm [10], which was formulated in the framework of single-agent search.

PN* transforms a best-first PN-search algorithm into an iterative-deepening depth-first approach. The PN* algorithm was implemented in a tsume-shogi (Japanese-chess mating-problem) program, and evaluated by testing it on 295 notoriously difficult tsume-shogi problems (one problem has a depth of search of over 1500 plies). The experimental results were compared with those of other programs. The PN* program showed by far the best results, solving all problems but one.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 157–168, 2008.

PDS [13], meaning Proof-number and Disproof-number Search, is a straight extension of PN* which uses only proof numbers. PDS is a depth-first algorithm using both proof numbers and disproof numbers. Therefore, PDS is basically more powerful than PN*. Moreover, Nagai [14] developed df-pn, meaning *depth-first proof-number search*. It behaves similarly to PN-search [15], but is more efficient in its use of memory. It solved all hard tsume-shogi problems quite efficiently.

Since then, df-pn has successfully been applied in other domains such as Go problems [9] and checkers [20,21]. However, we found a serious drawback when applying df-pn in other complex domains such as Othello. The drawback is known as the double-counting problem (see Subsection 2.3). Here we remark that Müller [12] calls it the problem of overestimation.

In this paper, we therefore explore a new idea to improve proof-number-based search algorithms and then propose a new search algorithm called *Weak Proof-Number Search*. We evaluate the new algorithm by testing it on some tsume-shogi problems and Othello endgame positions.

The contents of this paper are as follows. Section 2 presents a brief history of the development of proof-number-based AND/OR-tree search algorithms in the domain of mating search in shogi. Section 3 presents our new idea to improve the PN-search. Experimental performance in the domain of shogi and Othello are shown to evaluate the new search algorithm. Finally, concluding remarks are given in Sect. 4.

2 Proof-Number Based AND/OR-Tree Search Algorithms

Best-first algorithms are successfully transformed into depth-first algorithms, such as PN*, PDS, and df-pn. Each of these algorithms aimed at solving hard tsume-shogi problems [18]. The algorithms can be characterized as variants of proof-number search. Note that PN* only uses proof numbers, while PDS and df-pn use both proof numbers and disproof numbers. In this section, we give a short sketch of proof-number-based AND/OR-tree search algorithms.

2.1 PN-Search

The well-known technique of PN-search was designed for finding the game-theoretical value in game trees [2]. It is based on ideas derived from conspiracy-number search [11] and its variants, such as applied cn-search and $\alpha\beta$-cn search. While in cn-search the purpose is to continue searching until it is unlikely that the minimax value of the root will change, PN-search aims at proving the true value of the root. Therefore, PN-search does not consider interim minimax values. PN-search selects the next node to be expanded using two criteria: (1) the potential range of subtree values and (2) the number of nodes which must conspire to prove or disprove that range of potential values. These two criteria enable PN-search to deal efficiently with game trees with a non-uniform branching factor.

2.2 PN*, PDS, and df-pn

PN* [22] is a depth-first search using a transposition table, and a threshold for the proof numbers. PN* searches in a best-first manner, and uses much less working memory than the standard best-first searches.

Nagai proposed the PDS algorithm, that is, Proof-number and Disproof-number Search, which is a straight extension of PN* [13,14]. This search uses a threshold for the disproof number as well as the proof number when it selects and expands the nodes. The nodes with the smaller proof number or the smaller disproof number are searched first. If the proof number or the disproof number exceeds the threshold in a certain node, PDS stops further searching this node. When PDS fails to expand the root node, it increases one of the two threshold values and restarts the search.

PDS performs multiple iterative deepening in both AND nodes and OR nodes, while PN* does so only in OR nodes. Similarly to PN*, PDS's search behavior is in a best-first manner, while the search basically proceeds depth-first. From this point, PDS could be recognized as a variant of the proof-number search [2]. Actually, PDS uses proof and disproof number asymptotically while PN-search regards them fairly.

Nagai modified PDS and developed a new algorithm named df-pn [16]. The algorithm df-pn first sets the thresholds of both proof and disproof numbers in the root node to a certain large value that can be recognized as infinity. The threshold values are distributed among the descendant nodes. In every node, the multiple iterative deepening is performed in the same way as in PDS. Nagai [15] proved that df-pn search behaves in the same way as PN-search in the meaning that always a most-proving node will be expanded.

2.3 Possible Drawbacks of Proof-Number-Based Search Algorithms

PN-search has at least three possible bottlenecks [2]. The first is memory requirement. The second is Graph-History Interaction (GHI). The third is Directed Acyclic Graphs (DAGs). The answer to the first problem was df-pn, by which it became possible to solve efficiently quite difficult problems such as a shogi-mating problem with 1525 steps.

The GHI problem is a notorious problem that causes game-playing programs to return occasionally incorrect solutions [4]. PN-search and its depth-first variants also have to suffer from it. Breuker *et al.* [3] provided a solution for the best-first search. Later, Kishimoto and Müller [9] showed a practical method to cure the GHI problem for the case of the df-pn search.

A well-known problem of PN-search is that it does not handle transpositions very well. If the search builds a DAG instead of a tree, the same node can be counted more than once, leading to incorrect proof numbers and disproof numbers (i.e., *the double-counting problem*). Thus, PN-search overestimates proof numbers in case where DAGs occur. While there are exact methods for computing the proof numbers for DAGs [12], they are too slow to be practical. For some practical applications, Nagai [15] and Kakinoki [8] proposed a domain-dependent

improvement, respectively. However, the DAGs problem is still a critical issue when PN-search is applied in very hard domains.

2.4 AND/OR-Tree Search Taking Branching Factors into Account

A new search idea using the number of possibilities (i.e., branching factors) on a path considered as an estimator for AND/OR-tree search, instead of proof/disproof numbers, was proposed by Okabe [17]. It enables a solver to suffer relatively little from the serious problem due to DAGs. Experimental results show that for some very hard tsume-shogi problems with large DAGs, it outperforms df-pn. However, in most cases df-pn outperforms Okabe's search algorithm (named Branch Number Search or BNS in short).

Okabe [17] shows that in an example graph, threshold $n+1$ or more is needed to solve the graph by BNS, whereas 2^n or more is needed to solve the graph by df-pn. This indicates that for the number of n repeated DAGs a proof-number-based search algorithms suffer exponentially, whereas BNS suffers linearly.

Moreover, df-pn (with Nagai's improvement for DAGs) and BNS were compared in the domain of Othello [23]. It shows that as the frequency of DAGs grows, the performance of df-pn drastically decreases. Indeed, the frequency of DAGs increases as the number of search plies becomes larger in the domain of Othello. In the deeper search, therefore BNS outperforms df-pn in the execution time as well as in the number of search nodes.

3 Weak Proof-Number Search

In this section we propose a new search algorithm using information both on proof numbers and branching factors during search. First we present the basic idea of our proposed search algorithm. Then, the performance of the solver, in which the proposed idea is incorporated, is evaluated in the domain of tsume-shogi and Othello problems.

3.1 The Basic Idea of Our Proposed Search Algorithm

Our proposed idea is similar to PN-search. Hence, the implementation is easy. The only difference is, at an AND node, to use additional information (1) on branching factors and (2) on proof numbers. The information is used as a search indicator. In case where DAGs occur, the new search algorithm would better estimate the correct proof number than PN-search that often overestimates it.

The proposed search indicator, calculated as the maximum of the successor's proof number plus branching factors (except the maximum successor and solved/unsolvable successors) at an AND node, is somehow weaker (while underestimating it) than the proof number defined in PN-search. Therefore, we call it *Weak Proof-Number Search* or WPNS in short. The detail of the WPNS algorithm is shown in Appendix A. Note that procedure $\Phi Max(n)$ is its core part.

We expect WPNS to have two advantages: (1) when compared to proof-number-based search algorithms for relatively simple domains in which the DAGs

problem is not so critical and (2) when compared to the BNS algorithm for complex domains in which the DAGs problem occur frequently. Note that we usually have little knowledge about the DAGs issue for unknown target problems. Therefore, such synergy of proof number and branching factor for AND/OR-tree search would enable a program to be an all-round powerful solver.

In the case of a tsume-shogi problem [22], an OR node corresponds to a position in which the attacker is to move, where any move that solves the problem denotes a solution. The proof number then is the minimum proof number of its children (i.e., the one potentially easiest to solve). If the attacker has no more moves in the position, the problem is unsolvable from that position and the proof number is set to ∞.

Likewise, an AND node corresponds to a position with the defender to move. To solve the problem for the attacker all the defender's children must be proven to lead to the desired result, thus its proof number is the sum of the children's proof numbers. If the defender has no more legal moves (is mated), the goal is reached and the proof number is set to 0. However, as mentioned in Subsection 2.3, a serious double-counting problem will happen when large DAGs occur. Therefore, we propose a new search algorithm to use the weak proof number instead of Allis's proof number at AND nodes to avoid such a serious problem.

Let $p(n)$ denote the weak proof number of a node n in an AND/OR tree, and $d(n)$ denote weak disproof number. They are calculated as follows:

1. If n is a leaf node, then
   (a) if n is a terminal node and solved (i.e., OR wins), then
   $p(n) = 0$, $d(n) = \infty$;
   (b) else if n is a terminal and is unsolvable (i.e., OR does not win), then
   $p(n) = \infty$, $d(n) = 0$;
   (c) else n is an unsolved leaf node, then
   $p(n) = 1$, $d(n) = 1$;
2. else if n is an OR node whose successor nodes are $n_i (1 \leq i \leq K)$, then[1]
   $p(n) = \min_{1 \leq i \leq K} p(n_i)$,
   $d(n) = \max_{1 \leq i \leq K} d(n_i) + (k-1)$;[2]
3. else if n is an AND node whose successor nodes are $n_i (1 \leq i \leq K)$, then
   $p(n) = \max_{1 \leq i \leq K} p(n_i) + (k-1)$,
   $d(n) = \min_{1 \leq i \leq K} d(n_i)$.

For an easy-to-grasp example, see Fig. 1. In this example, the left-hand choice from the root node takes PN=17 and WPN=9 while the right-hand one takes PN=15 and WPN=11. This means that at the root node PN-search expands first the right-hand move whereas WPNS does the left-hand move. Indeed, the left-hand part is more plausible than the right-hand part in the sense of correct proof numbers. It happens because of a DAG in the left-hand part.

The example indicates that as the branching factor increases, the double-counting problem becomes more serious. We then argue that the performance of

[1] K is the number of successor nodes which do not have *terminal value* such as 0 or ∞.

[2] $(K-1)$ means the number of successor nodes which are not selected.

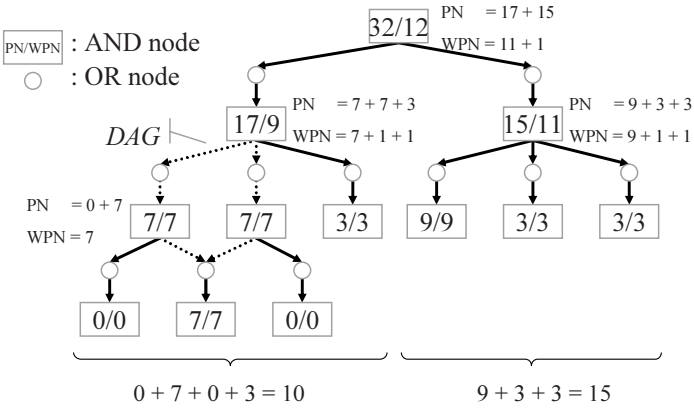


Fig. 1. Proof number (PN) and weak proof number (WPN) compared in an AND/OR tree with a DAG

PN-search decreases when solving a game with higher branching factors. However, the weak proof number is determined by the maximum proof number among all its successors at an AND node, and the number of remaining successors (i.e., branching factor -1) is added. Thus, WPNS relatively suffers little from the influence of DAGs.

3.2 Performance Evaluation

As mentioned in Sect. 2, proof-number-based search algorithms have remarkably been improved in the domain of tsume-shogi. Moreover, the double-counting problem of PN-search or its depth-first variants was found in the domain of Othello. Therefore, it is reasonable to use test sets from the two domains for a performance evaluation of the proposed idea.

WPNS, df-pn, and BNS in the Domain of Tsume-Shogi

In the first experiment, WPNS, df-pn, and BNS were compared in the domain of tsume-shogi. The machine environment was a 3.4 GHz Pentium4 PC running Windows XP and 2,000,000 entries of the transposition table used.

We selected a set of tsume-shogi problems from the book "Zoku-Tsumuya-Tsumazaruya" used as a suite of benchmark problems [7]. It contains 203 numbered problems (200 problems, with 1 problem subdivided into 4 subproblems) from the Edo era to the Showa era, created by 41 composers. The shortest problem is an 11-step problem and the longest one has a solution of 611 steps. The set contains various types of problems. Generally, the book is considered a good benchmark to measure the performance of a tsume-shogi solving program.

WPNS, df-pn, and BNS were implemented in Tacos that is a strong shogi-playing program [6], in which tsume-shogi specific enhancements such as non-pro-motion moves of major pieces are not incorporated. 113 problems were solved

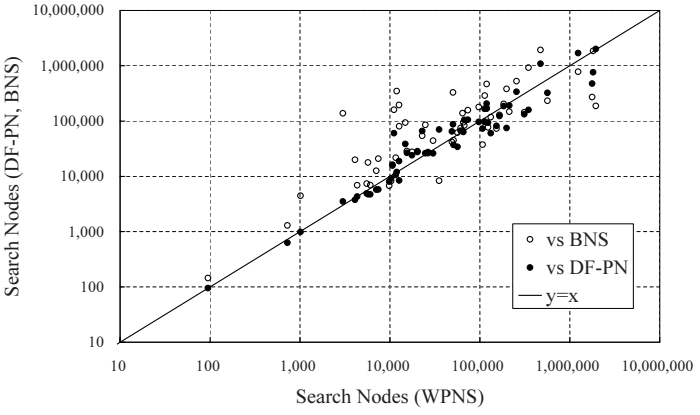


Fig. 2. WPNS, df-pn, and BNS compared on 59 tsume-shogi problems with short solutions

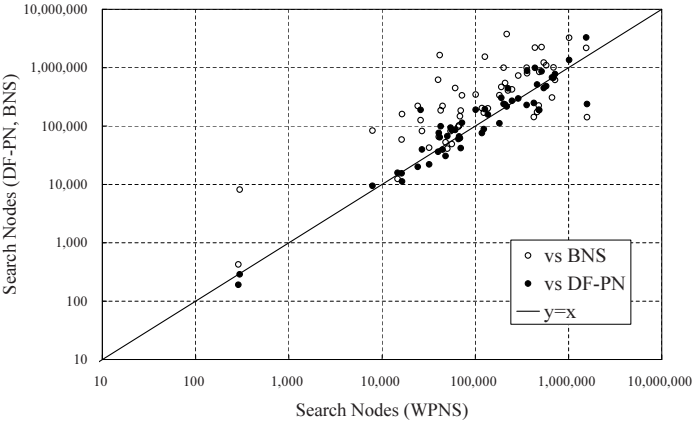


Fig. 3. WPNS, df-pn, and BNS compared on 54 tsume-shogi problems with relatively longer solutions

by each algorithm, whereas df-pn, BNS, and WPNS solved 131, 126, and 123 problems, respectively. We note that the problem with the longest solution was solved only by WPNS. The set of solved problems is categorized into two groups: 59 problems with short solutions (9 to 29 ply) and 54 problems with relatively longer solutions (31 to 79 ply). Let us show, in Fig. 2 and Fig. 3, the experimental results (i.e., the number of search nodes) on the first group and the second group, respectively.

For the first group df-pn outperforms WPNS by a small margin 2%, whereas for the second group WPNS outperforms df-pn by 6%. Moreover, for the first group WPNS outperforms BNS by 38%, whereas for the second group WPNS outperforms BNS by 66%.

WPNS, df-pn, and BNS in the Domain of Othello

In the second experiment, WPNS, df-pn, and BNS were compared in the domain of Othello. The machine environment was a 3.4 GHz Pentium 4 PC running Windows XP and 4,000,000 entries of the transposition table used. We obtained a set of Othello endgame positions through many self-play games using WZEBRA (Gunnar) [5], where each game started with a well-known opening position called *fjt1*. It contains 86 positions. The shortest problem has a 15-ply solution to end and the longest problem has a 20-ply solution.

Let us show, in Fig. 4, the experimental results. The results show that WPNS outperforms BNS by a large margin and is slightly better than df-pn. For 64 positions (75%), WPNS searched fewer nodes than df-pn, whereas for 84 positons (98%) WPNS searched fewer nodes than BNS.

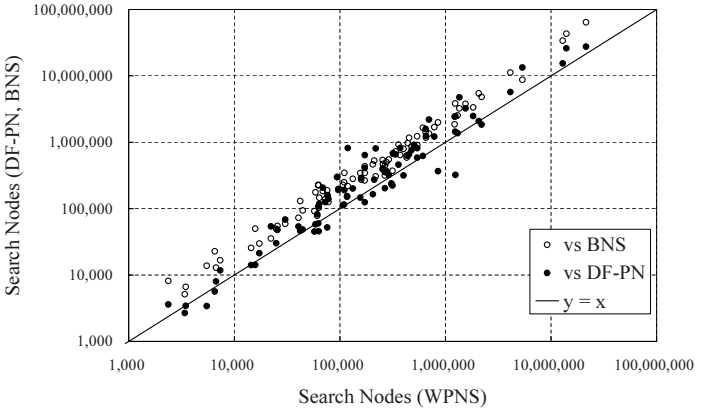


Fig. 4. WPNS, df-pn, and BNS compared on 86 Othello endgame positions

Discussion

From the experiments performed in the domains of tsume-shogi and Othello, we may observe that WPNS basically outperforms df-pn and BNS. In the context of the history of proof-number-based search algorithms, we may now state that df-pn is the most powerful solver in complex domains with some degree such as tsume-shogi problems with long solutions but relatively small branching factors. df-pn drastically decreases its performance in the domain of games with higher branching factors because of DAGs.

It is interesting to know that an average branching factor of tsume-shogi and Othello is 5 [22] and 10 [1], respectively. As shown in Fig. 1, the influence of double-counting problem due to DAGs becomes more serious in solving a game with higher branching factors as well as larger solutions. The experiments performed in the domain of tsume-shogi and Othello support our ideas. Therefore, we claim that WPNS outperforms df-pn when solving complex games with higher branching factors. The reason is that WPNS relatively suffers little from the influence of DAGs and df-pn suffers seriously.

4 Conclusion

Proof-Number Search (PNS) is a powerful AND/OR-tree search algorithm for efficiently solving a hard problem, although it has three possible bottlenecks (memory requirements, the GHI problem, the DAGs). The first two of them have already been improved. The last one is the double-counting problem discussed above. As we noticed in solving Othello endgame positions using df-pn this problem is the most notorious one when using a depth-first variant of PNS.

In this paper, we proposed a new AND/OR-tree search algorithm called Weak Proof-Number Search (WPNS). WPNS is a fruit of the synergy of PN-search and Branch Number Search (BNS). It takes an advantage of proof-number-based search algorithm and avoids the disadvantage of double-counting problem due to DAGs. Experiments performed in the domain of shogi and Othello show that WPNS can be a more powerful solver than df-pn.

References

1. Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, Computer Science Department, 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., van den Herik, H.J., Allis, L.V., Uiterwijk, J.W.H.M.: A Solution to the GHI Problem for Best-First Search. In: van den Herik, H.J., Iida, H. (eds.) CG 1998. LNCS, vol. 1558, pp. 25–49. Springer, Heidelberg (1999)
4. Campbell, M.: The graph-history interaction: on ignoring position history. In: 1985 Association for Computing Machinery Annual Conference, pp. 278–280 (1985)
5. Gunnar, A.: ZEBRA, http://radagast.se/othello/
6. Hashimoto, J.: Tacos wins Shogi Tournament. ICGA Journal 30(3), 164 (2007)
7. K. Kadowaki. Zoku-Tsumuya-Tsumazaruya, Shogi-Muso, Shogi-Zuko. Heibon-Sha, Toyo-Bunko, (1975) (in Japanese)
8. Kakinoki, Y.: A solution for the double-counting problem in shogi endgame. Technical report (2005) (in Japanese), http://homepage2.nifty.com/kakinoki_y/free/DoubleCount.pdf
9. Kishimoto, A., Müller, M.: Df-pn in Go: An Application to the One-Eye Problem. In: Advances in Computer Games 10, pp. 125–141. Kluwer Academic Publishers, Dordrecht (2003)
10. Korf, R.E.: Linear-space best-first search. Artificial Intelligence 62(1), 41–78 (1993)
11. McAllester, D.A.: Conspiracy numbers for min-max search. Artificial Intelligence 35(3), 287–310 (1988)
12. Müller, M.: Proof-Set Search. In: Schaeffer, J., Müller, M., Björnsson, Y. (eds.) CG 2002. LNCS, vol. 2883, pp. 88–107. Springer, Heidelberg (2003)
13. Nagai, A.: A new AND/OR tree search algorithm using proof number and disproof number. In: Proceedings of Complex Games Lab Workshop, pp. 40–45. ETL, Tsukuba (1998)
14. Nagai, A.: A new depth-first-search algorithm for AND/OR trees. M.Sc. Thesis, Department of Information Science, The University of Tokyo, Japan (1999)
15. Nagai, A.: Proof for the equivalence between some best-first algorithms and depth-first algorithms for AND/OR trees. In: Proceedings of Korea-Japan Joint Workshop on Algorithms and Computation, pp. 163–170 (1999)

16. Nagai, A., Imai, H.: Application of df-pn+ to Othello Endgames. In: Game Programming Workshop 1999, Hakone, Japan (1999)
17. Okabe, F.: About the Shogi problem solution figure using the number of course part branches. In: 10th Game Programming Workshop, Hakone, Japan (2005) (in Japanese)
18. Sakuta, M., Iida, H.: AND/OR-tree search algorithms in shogi mating search. ICGA Journal 24(4), 231–235 (2001)
19. Schaeffer, J.: Conspiracy numbers. In: Beal, D.F. (ed.) Advances in Computer Chess, vol. 5, pp. 199–218. Elsevier Science, Amsterdam (1989); Artificial Intelligence, 43(1):67-84 (1990)
20. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M., Lake, R., Lu, P., Sutphen, S.: Checkers Is Solved. Science 317(5844), 1518–1522 (2007)
21. Schaeffer, J.: Game Over: Black to Play and Draw in Checkers. ICGA Journal 30(4), 187–197 (2007)
22. Seo, M., Iida, H., Uiterwijk, J.W.H.M.: The PN*-search algorithm: Application to tsume-shogi. Artificial Intelligence 129(4), 253–277 (2001)
23. Ueda, T., Hashimoto, T., Hashimoto, J.: Solving an Opening Book of Othello and Consideration of Problem. In: 12th Game Programming Workshop, Hakone, Japan (2007) (in Japanese)

Appendix A

The C++ like pseudo-code of Depth-First Weak Proof-Number Search (DF-WPN) algorithm is given below. For ease of comparison we use similar pseudo-code as given in [14] for the df-pn algorithm. DF-WPN is similar to df-pn. The only difference is, at an AND node, to use additional information on branching factors as well as proof numbers, which appears in line 78.

Below code ϕ and δ are used instead of $WPN(n)$ and $WDN(n)$, just as α and β behave differently in the negamax algorithm compared to classical $\alpha\beta$ algorithm. These are defined as follows:

- $\phi = \begin{cases} WPN(n) & \text{if } n \text{ is OR node} \\ WDN(n) & \text{otherwise,} \end{cases}$
- $\delta = \begin{cases} WDN(n) & \text{if } n \text{ is OR node} \\ WPN(n) & \text{otherwise.} \end{cases}$

```
1  void df-wpn(root) {
2      root.thφ = ∞; root.thδ = ∞;
3      multiID(root);
4  }
5
6  void multiID(n) {
7      // 1. look up transposition table
8      retrieve(n, φ, δ);
9      if ( n.thφ ≤ φ || n.thδ ≤ δ ) {
10         return;
11     }
12     // 2. generate legal moves
```

```
13      if ( is_terminal(n)  ) {
14          if (( is_AND(n) && evaluate(n) = true  )||
15              ( is_OR(n)  && evaluate(n) = false )) {
16              store(n, ∞, 0); // cannot prove or disprove anymore
17          } else {
18              store(n, 0, ∞);
19          }
20          return;
21      }
22      generate_moves();
23      // 3. use transposition table to avoide cycle
24      store(n, φ, δ);
25      // 4. multiple iterative deepening
26      while (true) {
27          // stop if φ or δ is greater or equal to its threshold
28          φ = ΔMin(n);
29          δ = ΦMax(n);
30          if ( n.thφ ≤ φ || n.thδ ≤ δ ) {
31              store(n, φ, δ);
32              return;
33          }
34          child = select(n, φc, δc, δ2)
35          child.thφ = n.thδ + φc − δ;
36          child.thδ = min(n.thφ, δ2 + 1);
37          multiID(child);
38      }
39  }
40  // select the most proving child node
41  NODE select(n, &φc, &δc, &δ2) {
42      δc = ∞; δ2 = ∞;
43      for ( each child node c ) {
44          retrieve(c, φ, δ);
45          if ( δ < δc ) {
46              best = c;
47              δ2 = δc; φc = φ; δc = δ;
48          } else if ( δ < δ2 )
49              δ2 = δ;
50          if ( φ = ∞ )
51              return best;
52      }
53      return best;
54  }
55  // retrieve numbers from transposition table
56  void retrieve(n, &φ, &δ) {
57      if ( n is already recorded ) {
58          φ = Table[n].φ; δ = Table[n].δ;
59      } else {
60          φ = 1; δ = 1;
61      }
62  }
```

```
63 // store numbers to transposition table
64 void store(n, φ, δ) {
65     Table[n].φ = φ; Table[n].δ = δ;
66 }
67 // calculate minimum δ of the successors (same as df-pn)
68 unsigned int ΔMin(node) {
69     minδ = ∞;
70     for ( each child node c ) {
71         retrieve(c, φ, δ);
72         minδ = min(minδ, δ);
73     }
74     return min;
75 }
76 // calculate weak proof/disproof number
77 // df-pn uses ΦSum(n) instead of this function
78 unsigned int ΦMax(n) {
79     maxφ = 0;
80     for ( each child node c ) {
81         retrieve(c, φ, δ);
82         maxφ = max(maxφ, φ);
83     }
84     return ( maxφ + n.ChildNodeNum - 1 );
85 }
```

Cognitive Modeling of Knowledge-Guided Information Acquisition in Games

Reijer Grimbergen

Department of Informatics, Yamagata University, Yonezawa, Japan
grim@yz.yamagata-u.ac.jp

Abstract. Since Chase and Simon presented their influential paper on perception in chess in 1973, the use of chunks has become the subject of a number of studies into the cognitive behavior of human game players. However, the nature of chunks has remained elusive, and the reason for this lies in the lack of using a general cognitive theory to explain the nature of chunks. In this paper it will be argued that Marvin Minsky's *Society of Mind* theory is a good candidate for a cognitive theory to define chunks and to explain the relation between chunks and problem-solving tasks. To use Minsky's *Society of Mind* theory to model human cognitive behavior in games, we first need to understand more about the primitive agents dealing with the relation between perception and knowledge in memory. To investigate this relation, a reproduction experiment is performed in shogi showing that perception is guided by knowledge in long-term memory. From the results we may conclude that the primitive agents in a cognitive model for game-playing should represent abstract concepts such as *board*, *piece*, and *king* rather than the perceptual features of board and pieces.

1 Introduction

Game research has been a success story for the engineering approach, just like many other research areas in Artificial Intelligence. Deep Blue, probably the most famous of all game programs, searched between 100 million and 200 million positions per second in its 1997 match against Kasparov [3]. Human players clearly use a different approach, considering only a small number of positions per second and a small number of candidate moves (between 3 and 5) in any position [5].

In the past, there has been research by De Groot [5] into the behavior of chess players. Also well-known is the work by Chase and Simon [4], who introduced the idea of *chunking* of game knowledge to explain the difference between the performance of expert players and beginners in memory tasks. The nature of these chunks of game knowledge has been studied in other games such as Go [2,10], but there is not much known about chunks in games other than that they exist.

The most important reason for this omission is that there has never been an attempt to represent the essential game knowledge from the ground up. Without a proper understanding of how the most primitive building blocks of game

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 169–179, 2008.

knowledge interact to become chunks, it seems quite difficult to find the true nature of chunking. A general theory about human cognition is needed to define these building blocks and the interaction between them. Marvin Minsky's (1988) inspiring *Society of Mind* theory [9] is such a theory and our research aims at using Minsky's theory to simulate the chunking behavior of human game players.

In this paper the results of a reproduction experiment are given. They have important consequences for the content of the primitive agents and agencies dealing with input that are a vital part of a cognitive model for game-playing using Minsky's theory. Shogi (Japanese chess) will be used because we have performed earlier cognitive experiments in this game [6], but the results are general and do not depend on any shogi specific knowledge.

The rest of this paper is built up as follows. In Sect. 2 the theory behind the cognitive model for game-playing currently being built is explained. As a starting point, the primitive agents dealing with perception need to be defined. To investigate the nature of these primitive agents, in Sect. 3 a reproduction experiment will be described. The results of this experiment are given in Sect. 4. They show that perception in game playing is guided by game-specific knowledge and not by the perceptual features of the game. Finally, in Sect. 5 the conclusions and plans for future work are given.

2 A Cognitive Model for Perception in Games

To reproduce game positions, information about the positions must be stored in memory. Memory storage is often represented using the three-stage memory model proposed by Atkinson and Shiffrin [1]. This model states that human cognition is the result of the interaction between three different types of memory: sensory memory, short-term memory, and long-term memory (see Fig. 1). This three-way memory model is also the basis for the perception model for chess proposed by Simon and Chase [11], which will be partly followed.

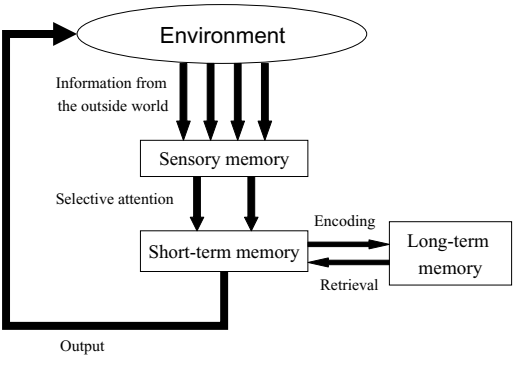


Fig. 1. Interaction between sensory memory, short-term memory, and long-term memory

Fig. 2. Perception guided by knowledge

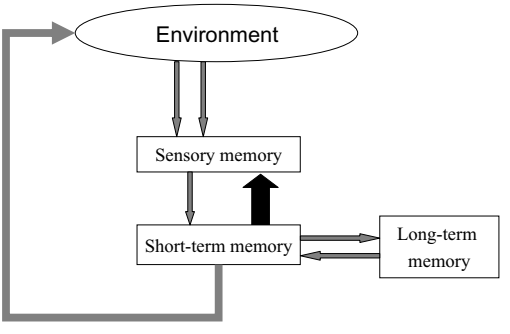


Fig. 3. Perception guided by knowledge in long-term memory

Sensory memory interacts with the environment by acquiring information through the senses. This is a subconscious process and therefore it cannot be guided. The amount of information that comes in through the senses is too high to process, so selective attention is used to limit the amount of information stored for further processing. This limited amount of storage is called short-term memory. Information in short-term memory can then be used to store and retrieve information from long-term memory or manipulate the environment.

Admittedly, this model of memory is too simplistic. However, it serves our modeling purposes except for one important extension. This is the phenomenon that we usually only see what we expect to see. For example, if we look at the picture of Fig. 2 for the first time, without any hints about what is in the picture, it is hard to see anything but a blur [7]. However, once we are told that the head of a cow is in the left side of the picture the blur changes into a cow. Furthermore, if we look at this picture again, we will find it very hard to "unsee" the cow.

The point of this example is to illustrate that perception seems to be guided by knowledge in long-term memory. Therefore, the actual three stage memory model used in our research is the one in Fig. 3, where knowledge from long-term memory is transferred to short-term memory. This information is often only

confirmed using sensory memory. In the example above, it means that by our knowledge that there is a cow in the picture (long-term memory knowledge), we just need to check that it is really there. When using this model, the task of short-term memory is threefold: (1) gathering information, (2) guiding environment interaction, and (3) confirming information.

Next, we explain how to build a cognitive model for games using the three-stage memory model given in Fig. 3 by looking at the features of each type of memory in more detail.

2.1 Sensory Memory

For each sense, there is a specific kind of sensory memory, but in games we only need to consider *iconic memory*, which is sensory memory dealing with visual stimuli. When we look at something, we fixate the central part of the eye (called the *fovea*). Such a fixation lasts from 200ms to up to 500ms or longer and the information gathered by a fixation is stored in iconic memory.

Experiments by Sperling [12] showed that iconic memory is like a snapshot picture. Even though most information is gathered around the point of fixation, we also have access to information further away. To avoid having to deal with all this information at once, selective attention is used to transfer a limited amount of information to short-term memory, where it can be used for further processing.

The content of iconic memory in games has been studied in detail by Tichomirov and Poznyanskaya [13]. They tracked the eye movement of an expert chess player during the first five seconds of trying to find the best move in a given position. They established that in these 5 seconds there were about 20 eye fixations. Most of these fixations were on squares occupied by pieces that could be considered important for that position. There were almost no fixations at the edges or corners of the board and also almost no fixations on empty squares. Furthermore, the fixations moved between pieces that could be considered to have a relation.

2.2 Short-Term Memory

Sperling's experiments also showed that the capacity of short-term memory is limited. The amount of information that can be stored was already known, because in 1956 Miller famously put a number on it: "The Magical Number Seven, Plus or Minus Two" [8]. Miller also gave the unit of this capacity the name *chunk*. A chunk is a piece of meaningful information, i.e., information that has a relation to information in long-term memory. A chunk can be quite small, like a single letter, but can also be much bigger. For example, a string of letters representing the name of a friend can be handled as a single chunk in short-term memory. As explained before, short-term memory has three different functions. Therefore, short-term memory is overwritten often and it is hard to measure exactly how long its storage capacity is. Estimates differ from 2 seconds to more than a minute.

A well-known study into the nature of chunks in games is performed by Chase and Simon [4]. They repeated earlier work by De Groot [5] in which chess players of different playing strength were asked to reproduce chess positions after viewing them for 5 seconds. The important difference with De Groot's work was that they also provided random positions. There were big differences in the reconstruction ability of normal chess positions, but the reconstruction ability was almost the same for random positions. The conclusion was that the difference in reproduction was caused by the fact that stronger players have bigger chunks of chess knowledge, so it is easier to fit a position having many pieces into the limited storage capacity of short-term memory.

Therefore, short-term memory can be modeled as a string of seven codes or link addresses to knowledge in long-term memory. The knowledge in long-term memory that is represented by this code can be very complex. The observation was already recognized by Simon and Chase [11] and implemented in their perception model. They went on and tried to simulate the behavior of the experts players from the Tichomirov and Poznyanskaya's experiments. However, this behavior turned out too complex, illustrated by the low number of eye fixations, indicating that a large amount of cognitive processing was involved. As a result, the Simon and Chase model was able to simulate some of the behavior observed by Tichomirov and Poznyanskaya, but failed to come up with a general framework for human cognition in games. Rather than making a model that tries to explain this complex behavior, it is better to start with the most basic behavior that is the same for players of all playing strengths. Therefore, the research presented here will first look in detail at the perception of board and pieces.

2.3 Long-Term Memory: The Society of Mind

Iconic memory and short-term memory are relatively well-understood, but this is not the case for long-term memory. The only thing that is certain is that the information in long-term memory lasts for decades and that its storage capacity is big enough to last a lifetime. Chase and Simon used so-called *inter-piece interval times* to investigate the nature of chunks in chess, but the jump from these inter-piece interval times, which are the same for players of different playing strength and chunks which are supposed to explain the differences between players of different playing strength is not convincing. Therefore, instead of following Simon and Chase, the nature of chunks will be investigated from the ground up.

Our approach for modeling long-term memory in games is to start with the most primitive chunks using Marvin Minsky's *Society of Mind* theory [9]. Minsky sees the mind as a large number of specialized cognitive processes, each performing some type of function. The simplest type of cognitive process is performed by an *agent* and the term *agency* is used to describe societies of agents that perform more complex functions.

Minsky defines an agent as: "Any part or process of the mind that by itself is simple enough to understand" [9]. It is important to realize that the cognitive processing units in the brain need to be simple, in the order of agents recognizing

color and shapes. Complicated behavior is the result of the interaction between groups of simple agents. Minsky describes a number of ways in which such an interaction can take place, the most important of which is the use of *K-lines*.

Minsky's theory is much more diverse than just agents, agencies, and K-lines. However, to use this theory for modeling game play, the first step is to understand the most primitive building blocks. Therefore, we first need to know about the agents that deal with input and output. The input for cognition in game-playing is perception of the board and pieces, while the output is playing moves. In the rest of this paper, it will be investigated how perceptual features of board and pieces influence the content of memory. Once we know this relation between perceptual features and cognition, the set of primitive agents can be decided. For example, if bigger pieces are more easily remembered than smaller pieces, we need an agency that can make a difference between pieces of different sizes.

To investigate the relation between perception and memory, a reproduction experiment has been carried out. This reproduction experiment will be described next.

3 Reproduction Experiment

To get a proper understanding of the fundamental agents dealing with perception, a reproduction experiment has been performed in the game of *shogi* (Japanese chess). Although the experiment has been done only for shogi, the same experiment can be done for any board game; the results are not expected to be game-specific. The main reason for this is that we made sure that no chunking was used. To achieve this, the reproduction experiment was performed using randomly generated shogi positions. Moreover, the subjects were all beginners at shogi, minimizing the amount of shogi-specific knowledge to guide perception using shogi chunks.

The experiment was designed to test the following four hypotheses.

Hypothesis 1: It is easier to perceive one's own pieces than the opponent's pieces. This hypothesis was based on the fact that in shogi (like in Chinese Chess), the name of the piece is written in Chinese characters on the piece. The Chinese characters of the opponent's pieces are thus seen upside down from the viewpoint of the player and might therefore be more difficult to perceive.

Hypothesis 2: It is easier to perceive promoted pieces than pieces that are not promoted. This hypothesis is based on the fact that the Chinese characters for promoted pieces are simpler than the characters for non-promoted pieces.

Hypothesis 3: Pieces closer to oneself are easier to perceive than pieces further away. This is the general perception principle of information about things near to oneself being more important than information about things that are further away.

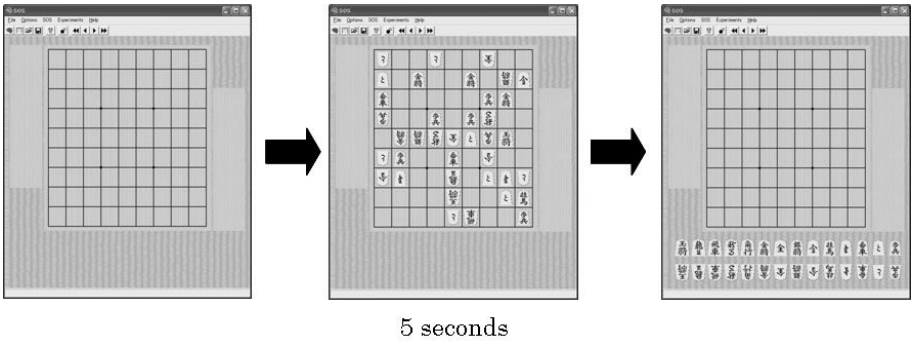


Fig. 4. Example of a position from the reproduction experiment

Hypothesis 4: Bigger pieces are easier to perceive than smaller pieces. This is also a general perception principle of bigger things being more important than smaller things.

The reproduction experiment to test these hypotheses was performed as follows (see Fig. 4). First, subjects were shown a shogi board without any pieces. When they felt ready to be shown the position, they pushed a button and a position would appear. This position would be shown for 5 seconds and then it would disappear, being replaced by an empty board with pieces lined up at the bottom of the screen. These pieces could then be moved to the board. There was no time limit for the reproduction phase of the positions. When the subjects felt that they had completed the task, they could click on a button and be shown the next position.

There were two positions used to explain the experiment and no data for these positions were recorded. In the experiment 10 randomly generated positions were used. The experiment is similar to the reproduction experiments we performed earlier [6], but with an important difference. The positions in our earlier experiments were generated by playing randomly from the starting position. Because of this, the generated position will have similarities with the well-known starting position, thus risking the use of chunks by the subjects.

We used 11 subjects in this experiment, all in their early twenties. Nine of the subjects had only a rudimentary knowledge of shogi, and two played a little more seriously in elementary school, but without ever gaining an official grade.

4 Experimental Results

Below, the results of the reproduction experiment related to the hypotheses will be presented.

Hypothesis 1: It is easier to perceive one's own pieces than the opponent's pieces. To test this hypothesis, data about the difference between the reproduction of own pieces (Chinese characters on the pieces displayed in the

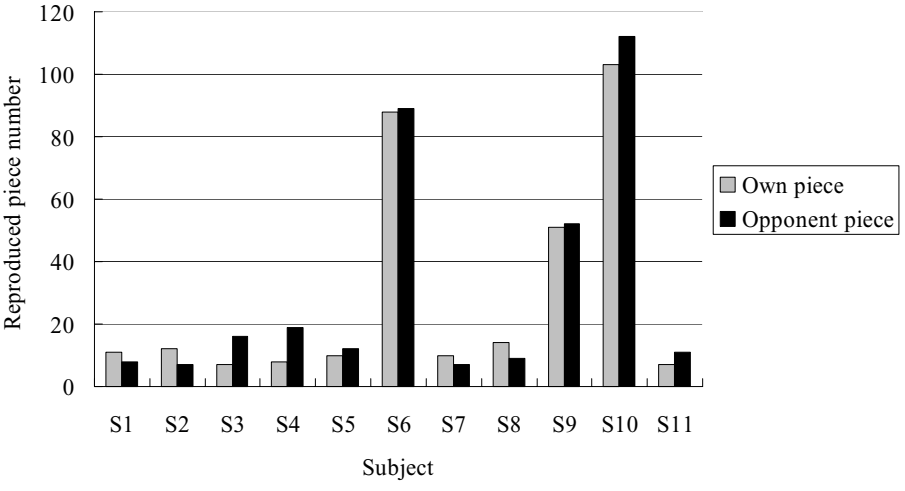


Fig. 5. Reproduction differences between own pieces and opponent pieces

normal way) and opponent pieces (Chinese characters displayed in reverse) was collected. The results are given in Fig. 5. From these results it can be concluded that in this experiment there was no data supporting the hypothesis. Only four subjects reproduced more of their own pieces than pieces of their opponent and only for subject S8 this difference seemed significant. Furthermore, the total number of own produced pieces was 321 (30.7%), while the total number of produced opponent pieces was 342 (31.7%).

Hypothesis 2: It is easier to perceive promoted pieces than pieces that are not promoted. To test this hypothesis, the difference between the reproduction of promoted pieces and non-promoted pieces was investigated. The results of this comparison are given in Fig. 6. From these results it can be concluded that non-promoted pieces are reproduced more than promoted pieces, so the hypothesis must be rejected. However, there are a number of subjects (S2, S3 and S11), who made an effort reproducing promoted pieces instead of non-promoted pieces. This did not lead to better performance regarding the correctness of the reproduced pieces, so this strategy seems to have no positive effect on memory storage.

Hypothesis 3: Pieces closer to oneself are easier to perceive than pieces further away. To test this hypothesis, a definition of *nearness* is needed. In the experiment, nearness is defined as the rank of the piece on which a piece is placed. The nearest pieces are therefore the pieces placed on the bottom rank, i.e., the rank closest to the player. Each rank further away is considered to be decreasing the nearness of the pieces. This assumption is consistent with the normal way of sitting behind a board. The results of piece reproduction according to this definition of nearness are given in Fig. 7. From this graph it is clear that there is no obvious relation between nearness and the reproduced

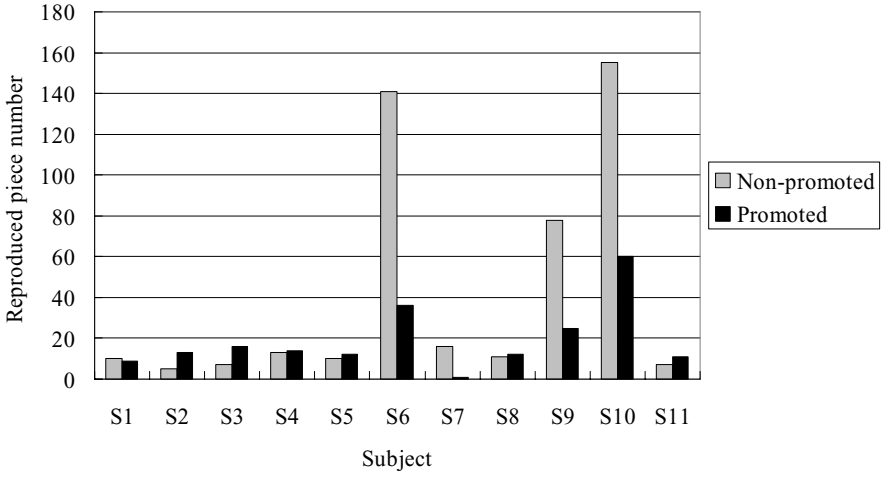


Fig. 6. Reproduction differences between promoted pieces and non-promoted pieces

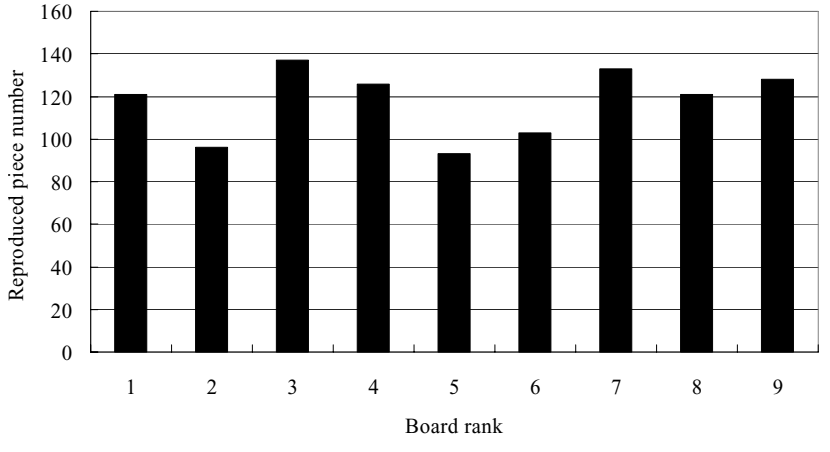


Fig. 7. Comparison of piece reproduction and nearness. Rank 1 represents the rank of the board closest to the subjects.

pieces and the hypothesis must therefore be rejected. In this case there was one subject who seemed to use a memorizing strategy where nearness played a role, but this subject reproduced pieces that were furthest away first, contradicting the assumption in the hypothesis.

Hypothesis 4: Bigger pieces are easier to perceive than smaller pieces. To test this hypothesis, data about the differences between the piece types of the reproduced pieces was collected. The standard relative size of each piece is given in Table 1. The pieces in the positions used in the experiment have the same relative piece size.

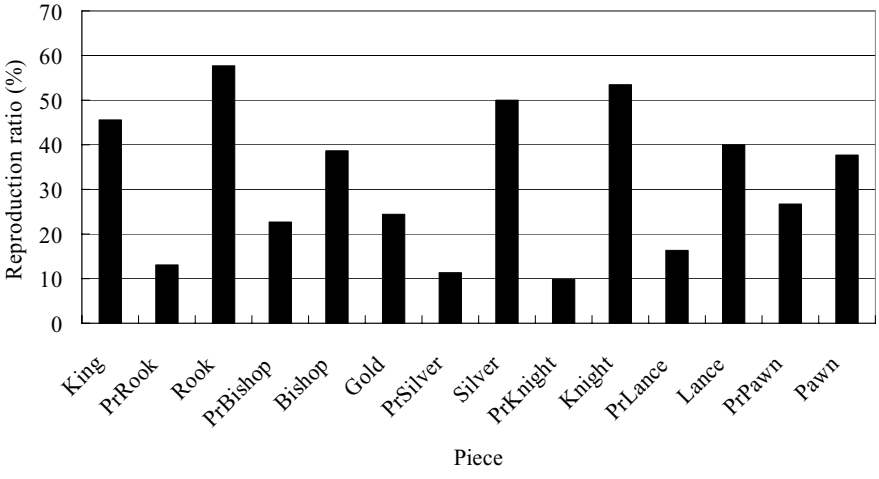


Fig. 8. Number of reproduced pieces for each piece type

Table 1. Piece sizes of shogi pieces in percentages relative to the size of the king. Note: promoted pieces have the same size as their non-promoted versions.

Piece	*RelSize*	*Piece*	*RelSize*
King	100	Silver	79
Rook	90	Knight	69
Bishop	90	Lance	59
Gold	79	Pawn	53

According to this table, the king should be reproduced more than the (promoted) rook and (promoted) bishop, which should in turn be reproduced more than gold and (promoted) silver, followed by (promoted) knight, (promoted) lance, and (promoted) pawn. The results of reproduction by piece type are given in Fig. 8. From this graph it may be concluded that there is no relation between reproduction ratio and piece size. Therefore, this hypothesis must also be rejected.

5 Conclusions and Future Work

In this paper it was explained why Marvin Minsky's Society of Mind theory is a good candidate for representing game-related knowledge in long-term memory. The goal of our research is to make a cognitive model for game-playing based on the Society of Mind theory. As a first step, in this paper a reproduction experiment was presented to get a proper understanding about the nature of the most primitive agents in the model, namely the agents that deal with the perception of board and pieces.

The experiment showed that perceptual clues in board and pieces (such as piece size) do not guide the knowledge stored in memory. This supports the

assumption that perception is guided by knowledge in long-term memory and that perceptual clues are only used to trigger this knowledge. From these results we may conclude that the primitive agents in our model do not need to represent perceptual features directly. Rather, agents and agencies can be built around primitive concepts such as *board*, *piece*, *king* and so on.

The next step is now to define the primitive concepts in a game and build agents, agencies, and the K-lines between them to represent these primitive concepts. This will require further reproduction experiments using beginners where the task has to be changed from reproduction to finding the primitive concepts among non-related information. For example, finding the king in a randomly generated position. Also, the concepts (chunks) used by players of different playing strength need to be investigated in order to understand how agents and agencies develop over time.

References

1. Atkinson, R.C., Shiffrin, R.M.: Human memory: A proposed system and its control processes. In: Spence, K.W., Spence, J.T. (eds.) The psychology of learning and motivation: Advances in research and theory, vol. 2, pp. 89–195. Academic Press, New York (1968)
2. Burmeister, J.: Memory Performance of Master Go Players. In: van den Herik, H.J., Iida, H. (eds.) Games in AI Research, Van Spijk, Venlo, The Netherlands, pp. 271–286 (2000)
3. Campbell, M., Hoane Jr., A.J., Hsu, F.-h.: Deep Blue. Artificial Intelligence 134(1–2), 57–83 (2002)
4. Chase, W.G., Simon, H.A.: Perception in chess. Cognitive Psychology 4, 55–81 (1973)
5. De Groot, A.D.: Thought and Choice in Chess. Mouton & Co, The Hague (1965)
6. Ito,T., Matsubara H., Grimbergen R.: Cognitive Science Approach to Shogi Playing Processes (1) – Some Results on Memory Experiments. Journal of the Information Processing Society of Japan, 43(10), 2998–3011, (2002) (in Japanese)
7. McCracken, D., Wolfe, R.: User-Centered Website Development. Prentice-Hall, New Jersey (2004)
8. Miller, G.A.: The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review 63(2), 81–97 (1956)
9. Minsky, M.: The Society of Mind. Simon and Schuster, New York (1988)
10. Reitman, J.: Skilled perception in Go: Deducing memory structures from inter-response times. Cognitive Psychology 8(3), 336–356 (1976)
11. Simon, H.A., Chase, W.G.: Skill in Chess. American Scientist 61, 394–403 (1973)
12. Sperling, G.: The Information Available in Brief Visual Presentations. Psychological Monographs: General and Applied 74(11, Whole No. 498), 1–29 (1960)
13. Tichomirov, G.K., Poznyanskaya, E.D.: An investigation of visual search as a means of analyzing heuristics. Soviet Psychology, 5, 2–15 (Winter 1966-1967)

Knowledge Inferencing on Chinese Chess Endgames

Bo-Nian Chen[1], Pangfeng Liu[1], Shun-Chin Hsu[2], and Tsan-sheng Hsu[3,⋆]

[1] Department of Computer Science and Information Engineering,
National Taiwan University, Taipei
{r92025,pangfeng}@csie.ntu.edu.tw
[2] Department of Information Management,
Chang Jung Christian University, Tainan
schsu@mail.cjcu.edu.tw
[3] Institute of Information Science, Academia Sinica, Taipei
tshsu@iis.sinica.edu.tw

Abstract. Several Chinese chess programs exhibit grandmaster playing skills in the opening and middle game. However, in the endgame phase, the programs only apply ordinal search algorithms; hence, they usually cannot exchange pieces correctly. Some researchers use retrograde algorithms to solve endgames with a limited number of attack pieces, but this approach is often not practical in a real tournament. In a grandmaster game, the players typically perform a sequence of material exchanges between the middle game and the endgame, so computer programs can be useful. However, there are about 185 million possible combinations of material in Chinese chess, and many hard endgames are inconclusive even to human masters. To resolve this problem, we propose a novel strategy that applies a knowledge-inferencing algorithm on a sufficiently small database to determine whether endgames with a certain combination of material are advantageous to a player. Our experimental results show that the performance of the algorithm is good and reliable. Therefore, building a large knowledge database of material combinations is recommended.

1 Introduction

Several Chinese chess programs are playing at a par with human masters or grandmasters [14]. Most algorithms that are incorporated in Western computer-chess programs are also suitable for Chinese chess programs. In the opening game, the most popular strategy involves building an opening book, either by collecting a large number of games or by inputting only master-level opening moves. The strategy is successful, in particular for general opening play. If a position is not in the book, the most important component, the search engine, takes over and computes the best move by evaluating hundreds of millions of positions.

⋆ Corresponding author.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 180–191, 2008.

Some programs can search more than 14 plies deep with today's computers. Although some computer-chess games end in the middle game, the endgame tends to be the key phase for strong programs.

However, in the endgame, the search performance is not comparable to the playing strength of master-level players. There are two reasons for this. The first reason is that players need more moves to finish the game than the search depth allotted to the program. The second reason is that the result of the endgame is not always related to the amount of material. For example, KR and KGGMM usually end in a draw, even though the former has the advantage of a rook. Hence, a program that uses the material advantage as the main evaluation feature often misinterprets it as a huge advantage to the attacking side.

To solve endgame problems, van den Herik and Herschberg suggested the concept of the retrograde strategy in 1985 [1]. Subsequently, van den Herik, Herschberg, and Nakad constructed a six-man endgame database of chess in 1987 [2]. Thompson proposed an improved retrograde algorithm in 1986 [6] and solved 6-piece chess endgames in 1996 [7]. Subsequently, Schaeffer (2003) created a 10-piece endgame database of Checkers [4]. Some games, like Checkers, used the retrograde method successfully [8]. For instance, Gasser solved Nine-Men's Morris in 1996 [11]. For the full game of Western chess, which is a complex game, the retrograde strategy has so far not been very successful. In 2000, Nalimov used an efficient partitioning of subgames to build all 3-to-5-men endgames [9]. In summary, we may state that the endgame research is still in progress.

In Chinese chess, Fang used the retrograde method to construct an endgame database in 2000 [3], and in 2002 Ren Wu [12] used a memory efficient strategy to build large endgames, including KGMCPKGGMM. In 2006, Wu, Liu, and Hsu proposed using an external-memory strategy for building a retrograde algorithm for a large endgame database [10]. Nowadays, there are also web sites that provide the exact values of endgame databases [5]. However, there are serious time and space limitations when constructing a practical endgame database of materials with sufficient attack pieces. The current largest endgame database of Chinese chess comprises no more than two strong attack pieces on each side. We remark that many useful endgames that contain two strong attack pieces on both sides cannot be solved by retrograde strategies.

In a typical grandmaster game, before a grandmaster applies his[1] endgame knowledge, he usually performs a series of material exchanges at the end of the middle game. In each material exchange, he gradually obtains an advantage. The advantage may not derive from accumulating more materials, but from a combination of materials that has proven to be better based on prior experiences. For example, it is generally believed that a combination of one rook, one horse, and one cannon is better than a combination of two horses and two cannons, although their material values are roughly equal. The goal of this paper is to determine whether a material combination is good by performing knowledge inferencing on a small dataset of kernel knowledge. To this end, we define two phases in the endgame: (1) *the prior phase*, during which many attack pieces are

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

still in position and retrograde strategies cannot be applied to them; and (2) *the posterior phase*, which can be solved completely by retrograde algorithms.

In particular, we propose a novel strategy that applies a knowledge-inferencing mechanism on a small knowledge database of material combinations to generate a database of material for the prior phase of a practical endgame.

The remainder of this paper is organized as follows. In Sect. 2, we describe the knowledge database of material combinations and the implemented knowledge-inferencing technique. In Sect. 3, we introduce a probabilistic model for predicting unknown material states. In Sect. 4, we build a practical knowledge database of material combinations. In Sect. 5, we take the data used by CONTEMPLATION [15] as our experimental data and report the results of applying our model to it. Then, in Sect. 6, we present our conclusions.

2 Constructing a Knowledge Database

To construct a practical knowledge database of material combinations, henceforth called a *material database*, we first need to construct a basic database. Instead of adding all data manually, we utilize knowledge-inferencing techniques in the construction phase to reduce the workload and the time required for the task.

2.1 Knowledge Database of Material Combinations

The word *material* denotes all pieces that appear in a specific position in both Western and Chinese chess. The *material state* of a position is an evaluation measurement that only considers material in the given position, not with respect to different locations.

For simplicity, we assume that two players in an endgame play either the attacking role or the defending role. The attacking role, which is called the *attacking player*, is defined as the player that has more attack power than the player with the defending role, who is called the *defending player*. We define 5 categories of material states for a material combination.

WIN: The score when the attacking player usually wins.
EASY_WIN: The score when the attacking player wins in many cases, but draws in some cases.
CHANCE_WIN: The score that ends in a draw in most cases, but the attacking player wins in some cases.
HARD_WIN: The score when the attacking player seldom wins.
UNKNOWN: The score when the attack power of either side is strong enough to capture the king of the opposite side; hence information about the material is not very useful.

A knowledge database of material combinations consists of the defending materials that players use. Each item of defending material is mapped to an attack file that includes all possible attack materials. Attack material is defined as the pieces that belong to the attacking player. The possible number of materials held

by a player in Chinese chess can be computed by combinatorics as follows. First, there are 27 combinations of strong pieces, including rooks, horses, and cannons. Second, pawns are divided into three categories, as defined in Subsection 2.3. By using combinations with repetition of all possible numbers of pawns, we retrieve the combinations of all categories of pawns, which total 56. Third, there are 9 combinations of defending pieces, including guards and ministers. Totally, a player can have 13,608 ($= 27 \times 56 \times 9$) possible material combinations; and the total number of possible material combinations on both sides is 185 million.

We have designed two useful knowledge inferencing strategies. The first strategy, *redundant attacking material checking and elimination*, which is described in Subsection 2.2, can be applied when creating both the basic database and database queries. The second strategy, called *pawn inferencing*, can only be used when creating the basic database. It is described in Subsection 2.3.

2.2 Redundant Attacking Material Checking and Elimination

This knowledge-inferencing tool can find and remove all attack material that is not necessary. The idea is that if we already know a material state is a WIN, material states to which attack material is added by one or more pieces are also WIN states because the attacking player has a bigger advantage in the WIN state. Similarly, if a material state is a HARD_WIN, material states from which attack material is taken by one or more pieces are also at most HARD_WIN states.

By using this algorithm, we can eliminate redundant attack materials when creating the basic database. For database queries, the same concept is used when there are some gaps between the attack power of two players. If the state of attack material found in the database is a WIN and the material is a subset of the query attack material, we can also report a WIN state. We call this inferencing algorithm *material state extension*.

A knowledge database of material combinations is said to be complete if all the database items that record defense materials have all the necessary information about attack materials. Generally, the time complexity of a query is $O(NM)$, where N is the number of defending materials in the database, and M is the maximum number of attacking materials among all defending materials in the database. However, if we use a complete material database, we do not need to search the whole database for the answer to a query. Instead, we only search the desired attacking file so that the time complexity becomes $O(M)$. We remark that the time so saved leads to more computation when searching.

2.3 Pawn Inferencing

In Chinese chess, as in Western chess, it is illegal to move a pawn backwards, but in Chinese chess a pawn is not promoted when reaching the final rank. Since the opposite player's king can only stay somewhere in the last 3 ranks, the distance between a pawn and the final rank decides the pawn's power. In the common definition, there are three types of pawns:

1. Top-pawn: the pawn stays behind the river line or the pawn line of the opposite side and has yet to cross the river. It moves forward 3 steps at most.
2. Low-pawn: the pawn moves forward 4 or 5 steps.
3. Bottom-pawn: the pawn reaches the final rank. Note that a pawn must move forward 6 steps to reach the final rank.

In general, a top-pawn is more useful than a low-pawn and a low-pawn is more powerful than a bottom-pawn. Furthermore, if we know the state of material with one bottom-pawn, we cannot obtain a better result by adding more bottom-pawns in all cases.

There is a similar rule for low-pawns. If we know the state of material with two low-pawns, we cannot obtain a better result by adding more low-pawns in most cases. There are two possible reasons for this. First, if low-pawns can win, then, based on past experience, only two low-pawns are sufficient to win. Second, if low-pawns cannot move into the palace or are lower than the king, adding low-pawns will not solve the problem. For example, the results of the material combinations KPPKGGMM and KPPPKGGMM are a CHANCE_WIN when all pawns are low-pawns. In our basic database, there are 16,705 material combinations where the attacking player has two low-pawns, and there are only 361 combinations where the result of corresponding material with three low-pawns is different. However, when there is one top-pawn in the material, the attacking player can always gain an advantage by adding another top-pawn.

The pawn-inferencing algorithm is a game-specific inferencing scheme that is only suitable for Chinese chess. It uses the knowledge of bottom-pawns and low-pawns. If we have the result of material containing one bottom-pawn or two low-pawns, we can use the algorithm to copy the results to more bottom-pawns or low-pawns until the number of bottom-pawns plus low-pawns equals 5. The algorithm reduces the work involved in creating the basic database by almost half. This is because the combinations of materials with more than one bottom-pawn or more than two low-pawns that can be generated automatically are approximately equal to the combinations of materials with one bottom-pawn or less than or equal to two low-pawns.

3 Predicting Unknown Material States

Although a large number of original unknown material states can be inferred by methods stated in Sect. 2, we still need a systematic strategy for handling arbitrary unknown materials. The algorithm that predicts arbitrary unknown materials is called the *unknown state predictor*.

3.1 Human Evaluation of Unknown Positions

By exchanging pieces, human experts can accurately infer the results of material combinations that were previously unheard of. For example, KHKGGMM is generally a draw. When the result of the material combination KRHKRGGMM

is in question, we may see the following: if the defending player has a rook, he can exchange it directly with the rook of the attacking player, and the result will be a draw. This strategy is called *material reduction*.

A second example is the material combination KRPKHGGMM. If the attacking side exchanges a pawn for two guards of the defending player, the resulting material KRKHMM can win easily, but it would not be an absolute win. However, if the pawn is exchanged for two ministers of the defending player, the resulting material, KRKHGG would be an absolute win.

The two examples show that making a correct exchange of pieces is important during the endgame phase.

3.2 Material Exchange Table

We have designed a probabilistic model that predicts the results of unknown material states by exchanging pieces. Both sides can exchange pieces when necessary. A material exchange table is introduced to compute the probabilities of exchanging pieces.

The mobility of many types of pieces is different. The ability to exchange a certain piece for pieces of another type is also different. A *helper piece* can be any piece that is not being exchanged, but it can be used to facilitate an exchange. Each player can select one piece as the helper piece. Generally, actively exchanging pieces with the assistance of a helper piece will increase the player's exchange ability. Similarly, passively exchanging pieces with the aid of a helper piece may reduce the chance of pieces being exchanged. Hence, we manually construct a two-dimensional material exchange table to record the probabilities of exchanging each type of piece with the assistance of helper pieces.

There are 6 types of pieces in addition to the king. To map a table to each active/passive piece pair, we use 36 tables for all possible types. Each table contains the probabilities of the specified active piece with all possible helper pieces and the specified passive piece with all possible helper pieces.

3.3 Determining the Score of an Unknown Material State

For an unknown material combination, we can try to make any exchange and to make a reference to the database for the material state. The strategy of an expert player is to choose the possible best way to make an exchange. We can accept an exchange that has a high probability, but we cannot accept an exchange with a low probability.

An *acceptable exchange* is formally defined as an exchange of which the material state is the most advantageous to the active player in all feasible situations and of which the probability is higher or equal to a lower bound. To achieve an acceptable probability of exchange and to avoid wasting time on searching for exchanges with low probability, we define the probability lower bound, PLB, to filter out situations with very low probability that seldom occur in practice. In our test, the best value of PLB is 10%. After an exchange, we make a reference to the database to retrieve the result of the reduced material. If two or more

exchanges result in the same material state, we choose the one with the highest probability. If we cannot find the result in the database, the material state of the specified material remains UNKNOWN.

The algorithm computes two acceptable exchanges: (1) the attacking player exchanges pieces actively, and (2) the defending player exchanges pieces actively. Each exchange reaches its own material state. We define five numerical score values, 0, 1, 2, 3, and 4, which correspond to UNKNOWN, WIN, EASY_WIN, CHANCE_WIN, and HARD_WIN, respectively. If the material states of both sides are known, the final score of the query material is computed by the formula $V = \lfloor (V_a + V_d)/2 \rfloor$. The values V_a and V_d represent the results of the attacking player and the defending player exchanging pieces actively, respectively. V is the final score. If one of the material states is unknown, we choose the known state as our result. If both are unknown, the result remains unknown. This formula simply computes the average of the two results. It is worth noting that, because we use division on integers, the result leans towards WIN rather than HARD_WIN, due to the setting of the numerical scores.

4 Constructing a Practical Knowledge Database of Material Combinations

We use two algorithms, material state extension and unknown state predictor, to determine the advantage of unknown materials.

To construct a knowledge database of material combinations, we simply generate each material pair as input for the material state extension algorithm, which can only be applied to WIN and HARD_WIN in the basic database. If the algorithm cannot find the answer, we input the material pair to the unknown state predictor algorithm to retrieve an approximate result value.

However, the value of some materials may still be unknown after applying the unknown state predictor algorithm. Finally, we use a heuristic algorithm to identify the advantage or disadvantage of the input material. We compute a player's attack power by the formula $10 \times Rook + 5 \times (Horse + Cannon) + 1 \times Pawn$. In the formula, Rook, Horse, Cannon, and Pawn are the numbers of the attacking pieces. The *difference between the attack power* of the two players is calculated as the formula $D = RedPower - BlackPower$. RedPower is the attack power of the attacking player, and BlackPower is that of the defending player. When D is more than or equal to 10, we reduce the value of the material state by one. When D is less than 7 and the predicted result is UNKNOWN, we set it to be CHANCE_WIN. This simple algorithm is used to fine tune the materials when the attacking player has a clear advantage or the value of the materials cannot be derived by the unknown state predictor algorithm.

The most practical usage of the knowledge database of material combinations is to retrieve material scores as a part of the evaluation function during the search phase. When a middle game position changes to an endgame position due to piece exchange, the search algorithm can select better endgame positions with the aid of our material database. However, there may be some positions

where the attack power of both sides is strong; or one player is disadvantaged in terms of material, but still represents a great threat to the opposite player's king. The former can be handled by assigning UNKNOWN states to the positions when both sides are strong enough to attack each other's kings. The latter can be handled by increasing the weight of special locations of piece combinations in the evaluation function.

5 Experiment Design and Results

To demonstrate the performance of our algorithm, we generate a basic database. It is a complete database of defense materials with at most one strong attacking piece plus one pawn and all defending pieces. We use a practical data set as our test data and compare it with the results obtained by our algorithm.

5.1 Experiment Design

We use the endgame knowledge table used by CONTEMPLATION as our test data. There are 17,038 combinations of materials that have been manually annotated by a 4-Dan expert. Since the data is symmetric, that is, if a material combination is in the database, information about exchanges between the attacking player and the defending player is also in the database, the actual number of test data combinations is 8,519. The scoring scheme used by the test data is different to that of our method. The score of the test data is divided into 10 values. The values 0 and 1 are mapped to WIN in our method, which means the attacking player usually wins. The value 2 is mapped to EASY_WIN, 3 is mapped to CHANCE_WIN, 4 is mapped to UNKNOWN, and 5 is mapped to HARD_WIN. The values from 6 to 9 indicate that the attacking player changes places with the defending player. The value 6 is mapped to CHANCE_WIN; 7 is mapped to EASY_WIN; and 8 and 9 are mapped to WIN. A second difference relates to the definition of pawns. In the test data, all pawns are the same, with no category information. As a result, our program must compute the approximate values of materials and then compare them with the test data. Because bottom-pawns are not considered by the test data, we only compute the approximate values of materials with top-pawns and low-pawns. The approximation formula is $V_{app} = \lfloor (V_{top} + V_{low})/2 \rfloor$, where V_{app} represents the approximated result; V_{top} represents the result of defining all pawns of both players as top-pawns; and V_{low} represents the result of replacing all pawns of the attacking player with low-pawns. There are 6,396 entries that are not in our basic database. We use the difference between the attack powers to filter out unreasonable annotations, which means that the attacking player has less attack power than the defending player, and is assigned the grade of better than or equal to CHANCE_WIN.

There are 1,621 annotations where the attacking player, who has the advantage of at least CHANCE_WIN, has less attack power than the defending player. The remaining data, containing 4,775 entries, becomes our test set, called END4775. We use two algorithms in the test: (1) material state extension, and

(2) unknown state predictor. The first experiment demonstrates the result of combining the two algorithms. The second experiment demonstrates the result of only using the unknown state predictor algorithm.

5.2 Experimental Results

In our results, we denote UNKNOWN by U, WIN by 1, EASY_WIN by 2, CHANCE_WIN by 3, and HARD_WIN by 4. The descriptions and results are shown in Table 2. We define the following variables to measure our model's performance: (1) *total_correct_number*, which records the number of cases where the output scores are equal to the transformed answer; (2) *tolerant_correct_number*, which ignores the error between WIN and EASY_WIN and also between CHANCE_WIN and HARD_WIN; and (3) *slight_error_number*, which records the errors between WIN and EASY_WIN and also between CHANCE_WIN and HARD_WIN.

For our algorithm, we need to choose a suitable value of PLB, described in Subsection 3.3. Table 1 shows the relationship between different PLBs and the ratio of tolerant_correct_number to the total number of data items, i.e., 4775. This is the most important measurement, when using only the unknown state predictor algorithm. As the results show, the value 10% is the best for our test data. We suggest that users set the PLB value between 10% and to 30%.

The total_correct_number is 2,169 or 45.42%. The tolerant_correct_number is 4,200 or 87.96%. The slight_error_number is 2,031 or 42.53%.

In practical usage, the most important measurement is tolerant_correct_number because it identifies the categories of either WIN and EASY_WIN, which are

Table 1. The relationships between PLBs from 0 to 100 and the corresponding ratio of tolerant_correct_number to the total number of data items

PLB	0	10	20	30	40	50	60	70	80	90	100
%	82.07	84.50	84.13	83.12	82.28	81.53	80.04	76.04	65.13	39.25	39.04

Table 2. Comparison of human annotated answers and the algorithm generated results for END4775. The horizontal axis represents the number of human annotated material states. The vertical axis represents the number of material states generated by the algorithm. U represents an unknown state.

	U	1	2	3	4	Sum
U	0	35	55	195	40	325
1	0	990	402	52	0	1444
2	0	1278	663	120	17	2078
3	0	31	30	233	330	624
4	0	0	0	21	283	304
Sum	0	2334	1150	621	670	4775

considered winning materials, or CHANCE_WIN and HARD_WIN, which are considered draw materials. The value 87.96% indicates the percentage of how well our algorithm fits a human expert's endgame knowledge. Moreover, the percentage shows the accuracy of the material part of the evaluation function used by a search algorithm in prior phase of the endgame, as defined in Sect. 1. Hence, the result shows that the search algorithm using our method will make as good an exchange as CONTEMPLATION in most cases.

A material combination KCPGGKPPP in our test data is assessed as EASY_WIN by a human expert, but reported as HARD_WIN by our algorithm. The discrepancy is due to the different opinions about the defense ability of a defending player who has three pawns. Since even masters have different opinions about hard endgames, a slightly different human annotated answer is reasonable.

The performance of the individual algorithms is as follows. The number of material combinations that can be inferred by the material state extension algorithm is 2,614. The total_correct_number among 2,614 entries is 1,379 (52.75%); the tolerant_correct_number is 2,562 (98.01%); and the slight_error_number is 1,183 (45.25%).

By using the heuristic strategy described in Sect. 4, we did not obtain any unknown material states in this test. The number of the entries that could not be handled by the material state extension is 2,152. However, they can be predicted by our predictor algorithm or the heuristic algorithm. The total_correct_number is 781 (36.29%); the tolerant_correct_number is 1,814 (84.29%); and the value of slight_error_number is 1,033 (48.00%).

Although the ratio of total correctness is reduced by using the heuristic strategy compared to that of combining two algorithms, we believe that our predictor algorithm is reliable for the following three reasons. First, the input of 2,152 entries is the most complex data among all data sets. Second, the total correctness ratio shows that, for the given material, the algorithm can distinguish the true advantage or disadvantage in endgames. Third, even master players cannot clearly identify the difference between WIN and EASY_WIN and between CHANCE_WIN and HARD_WIN based only on information about the material. For example, Y. C. Xu, a Chinese chess grandmaster, gave his opinions about a practical endgame in his publication "YinChang Chess Road." He criticized his opponent, G. L. Wu, who is also a Chinese chess grandmaster [13].

The human annotated answer for the material combination KHCPKHCM in our test data is an EASY_WIN; however, our algorithm reports CHANCE_WIN, which has different advantage. If a situation like this occurred during a real game, a grandmaster would not usually exchange an attack piece with his opponent because any exchange would result in a draw. These kinds of material combination problems cannot be solved by reducing the amount of material.

To evaluate the performance of using the unknown state predictor algorithm alone, we apply it to all 4,775 material combinations. The detailed experimental results are presented in Table 3.

The total_correct_number is 2,110 (44.19%); the tolerant_correct_number is 4,035 (84.50%) and the slight_error_number is 1,925 (40.31%).

Table 3. Results using only the unknown state predictor algorithm

	U	1	2	3	4	Sum
U	0	76	30	188	31	325
1	0	1302	30	112	0	1444
2	0	1615	281	158	24	2078
3	0	35	80	261	248	624
4	0	4	2	32	266	304
Sum	0	3032	423	751	569	4775

Note that the ratios of the tolerant_correct_numbers to the total number of data items are similar among the two tests, and so do the slight error number values. This shows that the data input to the unknown state predictor algorithm in the first experiment is really hard. The difference between the ratio of the tolerant_correct_number to the total number of data items of the two experiments is 3.46%. This indicates that the advantage predicted by our predictor algorithm is still reliable, even for hard data.

The space used to store all defense materials up to one strong piece plus a pawn and all combinations of defense pieces and their corresponding attack materials is 2.44M bytes.

6 Conclusions

Endgame problems represent a difficult issue in both Western chess and Chinese chess. The largest Chinese chess endgame database built by a retrograde algorithm currently contains only two strong attack pieces on each side. However, the endgame results show that many strong attack pieces exist. We have designed a knowledge inferencing scheme to build a practical material database for the initial phase of the endgame. In addition, we use the material state extension algorithm and the unknown state predictor algorithm to construct endgames with many strong attack pieces. Our experimental results show that the performance of our algorithms is good and reliable. When predicting the advantage of a material combination with a large number of pieces, we may conclude from the results above that our material state extension algorithm is an effective approach. However, if the extension algorithm fails, the predictor algorithm takes over and reports an inferred solution. This strategy can be used to solve the problem when a complete knowledge database of an endgame with a large amount of material cannot be built using conventional computer methods, and only advantage information is required to know for the material state.

Acknowledgments. This research was partially supported by National Science Council, Grants 95-2221-E-001-004 and 96-2221-E-001-004.

References

1. van den Herik, H.J., Herschberg, I.S.: The construction of an omniscient endgame data base. ICCA Journal 8(2), 66–87 (1985)
2. van den Herik, H.J., Herschberg, I.S., Nakad, N.: A six-men-endgame database: KRP(a2)KbBP(a3). ICGA Journal 10(4), 163–180 (1987)
3. Fang, H.R., Hsu, T.S., Hsu, S.C.: Construction of Chinese chess endgame databases by retrograde analysis. In: Marsland, T., Frank, I. (eds.) CG 2001. LNCS, vol. 2063, pp. 96–114. Springer, New York (2000)
4. Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., Sutphen, S.: Building the checkers 10-piece endgame databases. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games: Many Games, Many Challenges, vol. 10, pp. 193–210. Kluwer Academic Publishers, Dordrecht (2003)
5. Pai, J.T.: Chinese Chess Endgame Databases Query System, http://lpforth.forthfreak.net/endgame.html
6. Thompson, K.: Retrograde analysis of certain endgames. ICCA Journal 9(3), 131–139 (1986)
7. Thompson, K.: 6-piece endgames. ICCA Journal 19(4), 215–226 (1996)
8. Lake, R., Schaeffer, J., Lu, P.: Solving large retrograde analysis problems using a network of workstations. In: Advances in Computer Chess 7, Maastricht, The Netherlands, pp. 135–162 (1994)
9. Nalimov, E.V., Haworth, G.M., Heinz, E.A.: Space-efficient indexing of endgame databases for chess. In: van den Herik, H.J., Monien, B. (eds.) Advances in Computer Chess 9, pp. 93–113 (2001)
10. Wu, P.S., Liu, P.Y., Hsu, T.S.: An external-memory retrograde analysis algorithm. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 145–160. Springer, Heidelberg (2006)
11. Gasser, R.: Solving nine men's morris. In: Nowakowski, R. (ed.) Games of No Chance. MSRI, vol. 29, pp. 101–113. Cambridge University Press, Cambridge (1996)
12. Wu, R., Beal, D.F.: Fast, Memory-Efficient Retrograde Algorithms. ICGA Journal 24(3), 147–159 (2001)
13. Xu, Y.C.: YinChang Chess Road, Special Column 44-45. Yan Chen Ti Yu Newspaper Office (1997)
14. Yen, S.J., Chen, J.C., Yang, T.N., Hsu, S.C.: Computer Chinese Chess. ICGA Journal 27(1), 3–18 (2004)
15. CONTEMPLATION, A Chinese chess program, http://www.grappa.univ-lille3.fr/icga/program.php?id=112

Learning Positional Features for Annotating Chess Games: A Case Study

Matej Guid, Martin Možina, Jana Krivec, Aleksander Sadikov, and Ivan Bratko

Artificial Intelligence Laboratory, Faculty of Computer and Information Science,
University of Ljubljana, Slovenia
matej.guid@fri.uni-lj.si

Abstract. By developing an intelligent computer system that will provide commentary of chess moves in a comprehensible, user-friendly, and instructive way, we are trying to use the power demonstrated by the current chess engines for tutoring chess and for annotating chess games. In this paper, we point out certain differences between the computer programs which are specialized for playing chess and our program which is aimed at providing quality commentary. Through a case study, we present an application of argument-based machine learning, which combines the techniques of machine learning and expert knowledge, to the construction of more complex positional features, in order to provide our annotating system with an ability to comment on various positional intricacies of positions in the game of chess.

1 Introduction

The ever stronger chess programs are dangerous opponents to human grandmasters - already surpassing them in many aspects. In spite of that, their capabilities to *explain* why certain moves are good or bad in a language understandable to humans are quite limited. So far, only little attention was paid to an automatic intelligent annotation of chess games and consequently the progress made in this field is negligible in comparison to the enormous progress in the power of chess engines, which we have witnessed in the last decades. The typical "commentary" in the form of best continuations and their numerical evaluations can hardly be of much help to the chess-player who would like to learn the important concepts that are hidden behind the suggested moves.

For several years, the scientific research in this field was limited only to chess endgames, and the demonstrated concepts all had a common weakness - the inability to extend annotations to the entire game of chess[1]. In 2006, we introduced a new approach, which utilizes the power demonstrated by the current chess engines to provide commentary of chess games during all the phases of the game. Moreover, besides the ability of annotating tactical positions, the new approach enables the program to comment on various strategic concepts in given positions (see Sadikov *et al.* [5]). The main idea is to use the chess engine's evaluation function's features to describe the changes in the position when a move

[1] An interested reader can find an overview of the related work in Sadikov *et al.* [5].

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 192–204, 2008.

(or a sequence of moves) is made. The elementary features can later be combined to form higher-level concepts understandable to humans. The descriptions can be used both for the purpose of tutoring chess by providing knowledge-based feedback to students, and for the purpose of annotating chess games.

As we intend to demonstrate, evaluation functions of the programs that are aimed to *play* chess successfully do not need to be aware of all the concepts that would otherwise be useful for commenting on chess games. Hence, introducing additional positional features to the evaluation function of our annotating software turned out to be important for obtaining comprehensible, user-friendly, and instructive annotations.

Introducing new knowledge into the evaluation function of the program requires knowledge elicitation from a chess expert. This proved to be a hard task. Computer researchers usually find it hard to elicit useful knowledge from human experts. This *knowledge acquisition bottleneck* occurs because the knowledge is intuitive. The experts are quite capable of using their domain knowledge, but they find it much harder to formalize it and describe it systematically. Overcoming the knowledge acquisition bottleneck was of crucial importance and also the key motivation for introducing machine learning to construct additional positional features. In this paper, we present an application of a recent approach to machine learning - Argument Based Machine Learning (ABML, [2]) - by a case study in learning the positional feature bad bishop in a form suitable for annotating chess games. Such features may not always be suitable for evaluation functions of chess-playing programs, where time spent on heuristic search is important (besides, appropriate weights of these features should be determined). Nevertheless, they could serve well for annotation purposes.

The paper is organized as follows. In Sect. 2, we point out certain differences between positional features of a program which is specialized for playing chess and the program which is intended for providing quality commentary. In Sect. 3, through a case study "*The Bad Bishop*", we describe the construction process of a more complex positional feature that enables our annotating software to detect and comment on bad bishops. Section 4 provides conclusions and describes our future work.

2 Positional Features for Annotating Chess Games

The programs that are aimed to *play* chess successfully do not need to be aware of all the concepts that would otherwise be useful for giving instructive annotations. There is always a dilemma how much knowledge to implement into evaluation functions of the programs in order to achieve best tournament performances. The more knowledge means less time for efficient search and vice versa. It is commonly known that some programs have more knowledgeable evaluation functions, while others rely more on efficient search algorithms that allow them to reach higher search depths.

To illustrate our points, we will introduce a concept of *the bad bishop*. Watson [6] gives the following definition as traditional one: "A bishop that is on the same

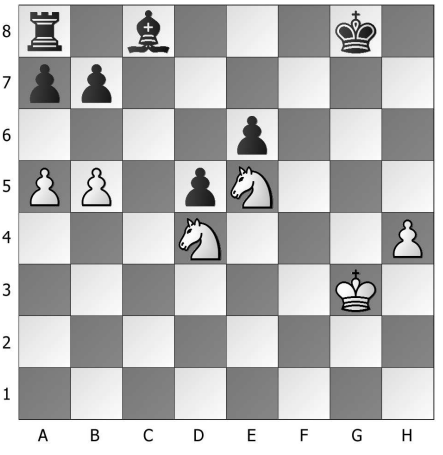


Fig. 1. Classical example of a bad bishop

color of squares as its own pawns is bad, since its mobility is restricted by its own pawns and it does not defend the squares in front of these pawns." Moreover, he puts forward that centralization of these pawns is the main factor in deciding whether the bishop is bad or not. In the middle game, he continues, the most important in this aspect are d and e pawns, followed by c and f pawns, while the rest of the pawns on the same color of a square as the bishop, are irrelevant (up to the endgame, where they might again become an important factor for determining the goodness of the bishop).

The example in Fig. 1 is taken from the classic book by Aaron Nimzovich, *The Blockade* [3]. The black bishop on c8 is bad, since its activity is significantly hindered by its own pawns. Furthermore, these pawns are blockaded by the pieces of his opponent, which makes it even harder for black to activate the bishop.

Looking at the chess programs, Crafty has several positional features that are associated with the goodness of the bishops. However, they are insufficient to fully describe this concept. They apply to both bishops for one side at the same time, i.e., the values for both bishops are represented by one feature only. Even if we arrange to obtain the feature values for each bishop separately, these positional features are still not appropriate to describe the goodness of a bishop, with the aim to annotate chess games in an instructive way.

Table 1 shows Crafty's most relevant positional features for describing a bad bishop. As it becomes clear from the descriptions of the features and their deficiencies from the purpose of describing bad bishops, Crafty clearly could not be aware of such a concept. For example, if we move pawns e6 and d5 to g6 and h7 (preserving the value of BLACK_BISHOP_PLUS_PAWNS_ON_COLOR - since pawns on the same color of the square as the bishop carry the same penalty, regardless of their position) and the rook from a8 to d7 (the value of BLACK_BISHOPS_MOBILITY even decreases, as the bishop is attacking one

Table 1. Some CRAFTY's positional features that have a potential for describing bad bishops and their deficiencies for doing so successfully

Feature	Description	Deficiency for annotating
BLACK_BISHOP _PLUS_PAWNS _ON_COLOR	a number of own pawns that are on the same color of the square as the bishop	all such pawns count the same, regardless of their position and how badly they restrict the bishop
BLACK_BISHOPS _POSITION	an evaluation of the bishop's position based on predefined values for particular squares	such predefined value is not the actual value of the bishop's placement in a particular position
BLACK_BISHOPS _MOBILITY	the number of squares that the bishop attacks	the number of attacked squares and the actual bishop's mobility are not necessarily the same thing

square less), the bishop clearly would not be bad, but in CRAFTY's view it would be even worse than in the given position.

In order to introduce a new positional feature (say BAD_BISHOP) that would allow commenting on such complex concepts, as is the concept of the bad bishop, it is therefore essential first to obtain additional (simple) positional features, and then combining them into some kind of rules that would allow to obtain the value of the new (more complex) positional feature BAD_BISHOP.

It should be noted that when annotating chess games, it is not necessary to comment on a particular feature each time it occurs. When the annotator is not sure about some feature in the position, it is better to say nothing at all than giving wrong comments.

2.1 The Static Nature of Positional Features

Positional features are static in their nature - they describe the state of their purposed issue for the current position only. It is heuristic search that enables them to fulfil their purpose - contributing to the program finding the best moves. For example, in the position in Fig. 1, if we moved the knight from e5 to h1, and decided that black is to move, black would easily solve all his problems by playing e6-e5, chasing the other white knight away and freeing both of his pieces. The positional features from Table 1, among others, would contribute to deciding for this freeing move, since the values of all three attributes become more desirable soon along the principal variation (e.g., after e6-e5 and Bc8-f5, there are less pawns on bishop's square color, and the bishop itself is placed on a square with a higher predefined value and also attacks more squares). Although the mentioned positional features are not suitable for commenting on the bad bishop, they nevertheless help it to become a good one.

It is also desirable for positional features for annotating chess games to be of static nature. For example, it is up to the chess engine to determine whether the freeing move e6-e5 is possible or not (e.g., in case of white king on f4 and the e5 knight still on h1 it would drop at least a pawn).

3 Case Study: The Bad Bishop

In this case study, we demonstrate the construction of a static positional feature, BAD_BISHOP (with possible values *yes* or *no*), which was designed for commenting on bad bishops (possibly combined with some heuristic search).

In our domain, it turns out to be extremely difficult for a chess expert to define appropriate rules, using typical positional features, for the program to be able to recognize complex concepts, such as *the bad bishop*. Our domain experts[2] defined the rules, using CRAFTY's positional features only, which in their opinion described bad bishops in the best possible way (considering the constraint of having only CRAFTY's features at disposal). The rules were of the following type:

```
IF (|BLACK_BISHOP_PLUS_PAWNS_ON_COLOR| > X)
AND (|BLACK_BISHOPS_MOBILITY| < Y) THEN BAD_BISHOP = yes
```

Three such rules were given, depending on the number of black pawns in the position. The positional features and the values for X and Y were determined, after the experts had become acquainted with the exact meaning of CRAFTY's positional features and had observed their values in several positions of various types. However, after examining the outcome of these rules on various chess positions, it turned out that the rules performed rather poorly, which was the key motivation for introducing machine learning into the system's development.

3.1 The Learning Dataset

The learning dataset consisted of middle game positions[3] from real chess games, where the black player has only one bishop. Based on the aforementioned expert-crafted rules, positions were obtained automatically from a large database of chess games. The bishops were a subject of evaluation by the experts.

When chess experts comment on concepts such as the bad bishop, they also have dynamic aspects of a position in mind. Therefore, assessing bishops "statically" is slightly counter-intuitive from the chess-player's point of view. After a careful deliberation, the following rules were chosen for determining a bad bishop from the static point of view.

The bishop is *bad* from the static point of view in some position, if:

1. Its improvement or exchange would notably change the evaluation of the position in favor of the player possessing it,
2. The pawn structure, especially the one of the player with this bishop, notably limits its chances for taking an active part in the game,
3. Its mobility in this position is limited or not important for the evaluation.

[2] The chess expertise was provided by WGM Jana Krivec and FM Matej Guid.

[3] While the concept of the bad bishop hardly applies to the early opening phase, different rules for determining bad bishops apply in the endgames (see Watson's definition in Sect. 2). In all positions, the quiescence criterion was satisfied.

These rules seem to be in line with the traditional definition of the bad bishop, and in the experts' opinion lead to sensible classification. In positions where assessment from the static point of view differs from the one obtained from the usual (dynamic) point of view, it seems likely that a possible implementation of heuristic search, would lead to sensible judgment on whether to comment on the bad bishop or not. In the implementation we would use the newly obtained positional feature BAD_BISHOP (such search could enable the program to realize whether the bishop is more than just temporarily bad and thus worth commenting on).

The learning dataset consisted of 200 positions[4]. We deliberately included notably more bishops labeled as "bad" by the initial rules given by the experts, due to our expectations (based on the unsuitability of CRAFTY's positional features) that many of the bishops labeled as "bad" will not be assessed so, after the experts' examination of the positions. After examination by the experts, 80 examples in the dataset obtained the class value *yes* ("bad") and 120 examples obtained the class value *no* ("not bad").[5] It turned out that only 59% of the examples were correctly classified by the expert-crafted rules.

3.2 Machine Learning

As the expert-crafted rules scored only 59% classification accuracy on our dataset, which is clearly insufficient for annotating purposes, there is a clear motivation for the use of machine learning. However, as classification accuracy equally penalizes false positives ("not bad" classified as "bad") and false negatives, we should also use precision, which measures the percentage of true "bad" bishops among ones that were classified as "bad". Remember, falsely commenting is worse than not commenting at all.

From the many available machine-learning methods we decided to take only those that produce understandable models, as it will be useful later to be able to give an explanation why a bishop is bad and not only labeling it as such. We chose standard machine-learning methods given in Table 2. We also give accuracy and precision results of these methods on our learning set.

Table 2. The machine learning methods' performance with CRAFTY's features

Method	Classification accuracy	Precision
Decision trees (C4.5)	71%	64%
Logistic regression	80%	76%
Rule learning (CN2)[6]	73%	75%

[4] This number was chosen as the most feasible one, considering limited available time of the experts. The quality of the final model implies that the number of selected positions in the learning dataset was sufficient.

[5] An interested reader will find the learning dataset, and some other remarkable details associated with this paper at the first author's website: http://www.ailab.si/matej.

All the accuracy and precision results were obtained through 10-fold cross validation. All the methods achieved better accuracies than the rules given by the experts, but the three methods are still too inaccurate for commenting purposes.

3.3 Argument Based Machine Learning

Argument Based Machine Learning (ABML, [2]) is machine learning extended with some concepts from argumentation. Argumentation is a branch of artificial intelligence that analyzes reasoning where arguments pro and con a certain claim are produced and evaluated [4].

Arguments are used in ABML to enhance learning examples. Each argument is attached to a single learning example only, while one example can have several arguments. There are two types of arguments; positive arguments are used to explain (or argue) why a certain learning example is in the class as given, and negative arguments are used to explain why it should not be in the class as given. We used only positive arguments in this work, as negatives were not required. Examples with attached arguments are called *argumented examples*.

Arguments are usually provided by domain experts who find it natural to articulate their knowledge in this manner. While it is generally accepted that giving domain knowledge usually poses a problem, in ABML they need to focus on one specific case only at a time and provide knowledge that seems relevant for this case and does not have to be valid for the whole domain. The idea can be easily illustrated with the task of commenting on chess games. It would be hard to talk about chess moves in general to decide precisely when they are good or bad. However, if an expert is asked to comment on a particular move in a given position, he or she will be able to offer an explanation and provide relevant elements of this position. Of course, in a new position the same argument could be incorrect.

An ABML method is required to induce a theory that uses given arguments to explain the examples. Thus, arguments constrain the combinatorial search among possible hypotheses, and also direct the search towards hypotheses that are more comprehensible in the light of an expert's background knowledge. If an ABML method is used on normal examples only (without arguments), then it should act the same as a normal machine-learning method. We used the method ABCN2 [2], an argument-based extension of the well-known method CN2 [1], that learns a set of unordered probabilistic rules from argumented examples. In ABCN2, the theory (a set of rules) is said to explain the examples using given arguments, when there exists at least one rule for each argumented example that contains at least one positive argument in the condition part[7].

In addition to rules, we need an inference mechanism to enable reasoning about new cases. Given the nature of the domain, we decided to learn only rules for "bad" bishop and classify a new example as "bad" whenever at least one of the learned rules triggered.

[7] Due to space limitations, we only roughly described some of the properties of ABML and ABCN2 (see [2] or/and the website http://www.ailab.si/martin/abml for precise details).

Asking experts to give arguments to the whole learning set is not likely to be feasible, since it requires too much time and effort. The following loop describes an iterative process for acquiring arguments and new attributes from experts:

1. Learn a set of rules.
2. Search for problematic cases in the data set; these are the examples that are misclassified by the induced rules.
3. If no problematic examples are found, stop the process.
4. Select a problematic example and present it to experts. If the case is a position with a "bad" bishop, then experts are asked to explain why this bishop is "bad". If it is a "not bad" bishop position, then we search for the culpable rule predicting "bad" and ask experts to explain an example with the class value *yes* ("bad") from the set of examples covered only by this rule. In the latter case, experts need to be careful to provide reasons that are not true in the problematic position. Problematic positions with a "not bad" bishop are called *counter-examples*.
5. Experts have three possibilities of responding to the presented case.
   (a) They can give reasons why the bishop is "bad". Reasons are added to the example in the data set.
   (b) If they cannot explain it with the given attributes, they may introduce a new attribute (or improve an existing one), which is then added to the domain.
   (c) Experts can decide that this bishop is actually "good" and thus the class of the example needs to be changed.

   If experts are unable to explain the example, we select another one.
6. Return to step 1.

Table 3 shows the three rules induced in the first iteration of the aforementioned process, where only CRAFTY's positional features were used and no arguments have been given yet. The condition part of a rule is the conjunction of the features indicated in the corresponding column. In the cases with no threshold specified, the feature is not part of the corresponding rule. For example, the rule #1 is: "IF BLACK_BISHOP_MOBILITY > -12 THEN BAD_BISHOP = yes".[8]

The distribution of positive and negative examples covered by each of the rules speaks about the relatively poor quality of these rules - in particular that of the last rule.

Figure 2 (left) shows the first problematic example selected by our algorithm. The experts were asked to describe why the black bishop is bad. Based on their answer, another attribute, BAD_PAWNS, was added into the domain. The experts designed a look-up table (right) with predefined values for the pawns that are on the color of the square of the bishop in order to assign weights to such

[8] The negative values of BLACK_BISHOP_MOBILITY are the consequence of CRAFTY using negative values for describing features that are good for Black. The more squares this bishop is attacking, the more negative this value. For each attacked square, the feature's value is decreased by -4. For example, the value of -12 means that the bishop is attacking three squares.

Table 3. The rules for BAD_BISHOP = *yes*, after the first iteration of the process

Positional feature	**#1**	**#2**	**#3**
BLACK_BISHOPS_MOBILITY	> -12	> -18	> -18
BISHOP_PLUS_PAWN_ON_COLOR			> 12
BLACK_BISHOP_POSITION		> 4	
positive examples ("bad")	40	44	67
negative examples ("not bad")	4	7	25

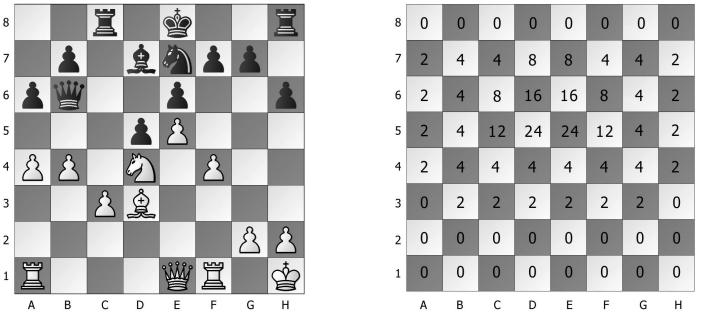


Fig. 2. The experts were asked the question: "*Why is the black bishop bad?*" They used their domain knowledge to provide the following answer: "*The black bishop is bad, since a lot of black pawns are on the same color as the bishop. Especially the central pawns notably limit its chances for taking an active part in the game.*" The need for the attribute BAD_PAWNS was identified. The experts designed a look-up table with predefined values for the pawns that are on the color of the square of the bishop in order to assign weights to such pawns.

pawns. According to Watson's definition, centralization of the pawns has been taken into account. Several other attributes that were added at later stages (see Table 4) used this look-up table to assess heuristically an influence of such bad pawns on the evaluation of the bishop.

Table 4 presents the list of attributes that were added to the domain during the process. To give an example of how the values of these new attributes are obtained, we calculate the value of the attribute BAD_PAWNS_AHEAD for the position in Fig. 2. This attribute provides an assessment of pawns on the same color of square as the bishop of which they are in front of. There are three such pawns in that position: e6, d5, and a6. For each of these pawns their corresponding values are obtained from the look-up table, that is, 16, 24, and 2, respectively. The sum of these values (16 + 24 + 2 = 42) represents the value of the attribute BAD_PAWNS_AHEAD in that position.

Figure 3 shows an example how the argument given to some particular position could be improved by the expert, using some help by a machine-learning method, which automatically suggests an appropriate counter-example. The

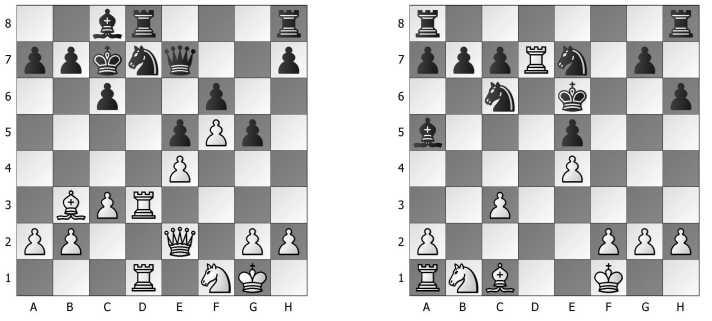


Fig. 3. After iteration 6, the expert gave the following description why the bishop is bad in position on the left: "*The bishop is bad, because, taking the pawn structure into account, only one square is accessible to it.*" The argument "IMPROVED_BISHOP_MOBILITY=low" was added to this position, based on this description. However, in the next iteration, the machine-learning method selected the position on the right, where the bishop is classified as "not bad", as the counter-example. After the expert's examination, the following significant difference between the two positions was determined: in the position on the right, there are no bad pawns ahead of the bishop. Based on that, the argument to the position on the left was improved to "IMPROVED_BISHOP_MOBILITY=low AND BAD_PAWNS_AHEAD=high".

Table 4. The new attributes, and iterations when they were added to the domain

Attribute	**Description**	**It.**
BAD_PAWNS	pawns on the color of the square of the bishop - weighted according to their squares (*bad pawns*)	2
BAD_PAWNS_AHEAD	bad pawns ahead of the bishop	3
BLOCKED_DIAGONAL	bad pawns that block the bishop's (front) diagonals	4
BLOCKED_BAD_PAWNS	bad pawns, blocked by opponent's pawns or pieces	5
IMPROVED_BISHOP _MOBILITY	number of squares accessible to the bishop, taking into account only pawns of both opponents	6
BLOCKED_PAWNS _BLOCK_DIAGONAL	bad pawns, blocked by opponent's pawns or pieces, that block the bishop's (front) diagonals	12

counter-examples are another effective feature for overcoming the knowledge acquisition bottleneck.

The final rules at the end of the process are presented in Table 5 (for the interpretation of this presentation, see the description before Table 3). The obtained rules for the new positional feature BAD_BISHOP only cover positive examples, have a pure distribution (no misclassified examples), and also appear sensible to the experts.

Particularly valuable is that the rules enable not only commenting on whether a bishop is bad, but also *why* it is bad. Formulation of the explanations is

Table 5. The rules for BAD_BISHOP = *yes*, obtained after the 14^{th} (final) iteration

Positional feature	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**
BAD_PAWNS					> 14		> 32
BAD_PAWNS_AHEAD	> 20	> 18		> 26	> 28	> 12	
BLOCKED_DIAGONAL	> 4		> 16				> 16
BLOCKED_BAD_PAWNS				> 0			
IMPROVED_BISHOP_MOBILITY		< 3	< 4	< 4		< 2	< 5
BLOCKED_PAWNS _BLOCK_DIAGONAL					> 0		
BLACK_BISHOPS_MOBILITY	< -15						
positive examples ("bad")	46	46	42	38	38	36	31
negative examples ("not bad")	0	0	0	0	0	0	0

Table 6. The machine-learning methods' performance with the data, supplemented by the newly obtained attribute values

Method	**Classification accuracy**	**Precision**
Decision trees (C4.5)	85%	85%
Logistic regression	89%	91%
Rule learning (CN2)	91%	94%
Rule learning with arguments (ABCN2)	94%	96%

provided in the expert module of our annotating system. For example, if the rule #2 from Table 5 triggers in a particular position, the following comment could be given: "*Black bishop is bad, since black pawns on the same color of squares ahead of it, and pawns of both opponents restrict its mobility.*"

The machine-learning methods that were used on original CRAFTY's positional feature values were again tested on the same data, supplemented by the newly obtained attribute values. All the accuracy and precision results were again obtained through 10-fold cross validation.

The results are presented in Table 6. They suggest that the performance of other algorithms could also be improved by adding appropriate additional attributes. However, using arguments (as with the method ABCN2), besides stimulating the expert to identify the need for useful additional attributes, also guides the method towards appropriate combinations of attributes, which is likely to lead to even more accurate models.

4 Summary and Conclusions

We investigated a particular aspect in the development of a chess annotating tool - the ability of making intelligent comments on the positional aspects of a chess game. This task is made more difficult by the fact that the strength

of the chess-playing programs mainly comes from search and not from subtle positional knowledge which is necessary for generating positional comments. Therefore, components of a chess program's evaluation function are not sufficient for making in-depth positional comments. Defining deep positional patterns requires powerful knowledge-elicitation methods. Our study suggests that argument-based machine learning enables such a method.

Our approach to the generation of positional comments makes use of elements of a chess evaluation function. However, more sophisticated positional patterns have to be introduced in addition to the features contained in an evaluation function. Defining such sophisticated positional patterns is often a difficult knowledge-elicitation task.

In the presented case study, we considered the elicitation of the well-known chess concept of the bad bishop. There is a general agreement in the chess literature and among chess players about the intuition behind this concept. However, formalizing it in a way that would enable an annotating system to decide reliably whether a bishop in a given position is bad turned out to be beyond the practical ability of our chess experts (a master and a woman grandmaster). The introduction of sophisticated positional concepts such as the bad bishop turned out to be an intricate knowledge-elicitation problem.

To alleviate the knowledge-elicitation problem, we employed machine learning from examples of good and bad bishops. We tried several standard machine-learning techniques to induce definitions of the bad bishop from examples. This did not produce satisfactory results. Finally, we successfully applied the recently developed approach of Argument Based Machine Learning (ABML). The efficacy of ABML comes from its unique ability to make use of expert's arguments (or justifications) for selected example cases. This approach is natural and effective for the expert because he or she can concentrate on explaining concrete cases, and does not have to construct general rules, which is more difficult. The method also leads the expert to think about new relevant descriptive features, thereby improving the description language that the learning program uses. In our case study, we may conclude that the method worked well. The initial repertoire of the attributes taken directly from Crafty's evaluation function was extended by another six attributes that the experts coined when explaining critical cases selected automatically by the ABML-based knowledge-elicitation process.

Our future work will be associated with further improvements of our annotation software. We intend to implement several additional positional features into its evaluation function, in order to make the commentary more instructive. In particular, the expert module of our annotation tool, which provides the user with a commentary of chess games, and possibly with more detailed explanations about particular features of chess positions, requires further attention. The tool will be based on both learned and manually crafted positional features. As part of future work, we intend to apply this knowledge-acquisition method to the formalization of other positional concepts of fuzzy nature, such as weak or strong pawn structures, "harmony among the pieces", etc.

References

1. Clark, P., Boswell, R.: Rule induction with CN2: Some recent improvements. In: Kodratoff, Y. (ed.) EWSL 1991. LNCS, vol. 482, pp. 151–163. Springer, Heidelberg (1991)
2. Možina, M., Žabkar, J., Bratko, I.: Argument based machine learning. Artificial Intelligence 171(10/15), 922–937 (2007)
3. Nimzovich, A.: The Blockade. Republished by Hardinge Simpole Limited (2006)
4. Prakken, H., Vreeswijk, G.: Handbook of Philosophical Logic. In: chapter Logics for Defeasible Argumentation, 2nd edn., vol. 4, pp. 218–319. Kluwer Academic Publishers, Dordrecht (2002)
5. Sadikov, A., Možina, M., Guid, M., Krivec, J., Bratko, I.: Automated chess tutor. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 13–25. Springer, Heidelberg (2006)
6. Watson, J.: Secrets of Modern Chess Strategy. Gambit Publications (1999)

Extended Null-Move Reductions

Omid David-Tabibi[1] and Nathan S. Netanyahu[1,2]

[1] Department of Computer Science, Bar-Ilan University,
Ramat-Gan 52900, Israel
mail@omiddavid.com, nathan@cs.biu.ac.il
[2] Center for Automation Research, University of Maryland,
College Park, MD 20742, USA
nathan@cfar.umd.edu

Abstract. In this paper we review the conventional versions of null-move pruning, and present our enhancements which allow for a deeper search with greater accuracy. While the conventional versions of null-move pruning use reduction values of $R \leq 3$, we use an aggressive reduction value of $R = 4$ within a verified adaptive configuration which maximizes the benefit from the more aggressive pruning, while limiting its tactical liabilities. Our experimental results using our grandmaster-level chess program, Falcon, show that our *null-move reductions* (NMR) outperform the conventional methods, with the tactical benefits of the deeper search dominating the deficiencies. Moreover, unlike standard null-move pruning, which fails badly in zugzwang positions, NMR is impervious to zugzwangs. Finally, the implementation of NMR in any program already using null-move pruning requires a modification of only a few lines of code.

1 Introduction

Chess programs trying to search the same way humans think by generating "plausible" moves dominated until the mid-1970s. By using extensive chess knowledge at each node, these programs selected a few moves which they considered plausible, and thus pruned large parts of the search tree. However, plausible-move generating programs had serious tactical shortcomings, and as soon as brute-force search programs such as Tech [17] and Chess 4.x [29] managed to reach depths of 5 plies and more, plausible-move generating programs frequently lost to brute-force searchers due to their tactical weaknesses. Brute-force searchers rapidly dominated the computer-chess field.

The introduction of null-move pruning [3,13,16] in the early 1990s marked the end of an era, as far as the domination of brute-force programs in computer chess is concerned. Unlike other forward-pruning methods (e.g., *razoring* [6], Gamma [23], and *marginal forward pruning* [28]), which had great tactical weaknesses, null-move pruning enabled programs to search more deeply with minor tactical risks. Forward-pruning programs frequently outsearched brute-force searchers, and started their own reign which has continued ever since; they have won all World Computer Chess Championships since 1992. Deep Blue [18,21]

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 205–216, 2008.

was probably the last brute-force searcher. Today almost all top-tournament playing programs use forward-pruning methods, null-move pruning being the most popular of them [14].

In this article we introduce our *extended null-move reductions*, and demonstrate empirically its improved performance in comparison to standard null-move pruning and its conventional variations. In Sect. 2 we review standard null-move pruning and its enhancements, and in Sect. 3 we introduce extended null-move reductions. Section 4 presents our experimental results, and Sect. 5 contains concluding remarks.

2 Standard Null-Move Pruning

As mentioned earlier, brute-force programs refrained from pruning any nodes in the full-width part of the search tree, deeming the risks of doing so as being too high. Null-move [3,13,16] introduced a new pruning scheme which based its cutoff decisions on dynamic criteria, and thus gained greater tactical strength in comparison with the static forward-pruning methods that were in use at that time.

Null-move pruning is based on the following assumption: in every chess position, doing nothing (i.e., doing a null move) would not be the best choice even if it were a legal option. In other words, the best move in any position is better than the null move. This idea enables us easily to obtain a lower bound α on the position by conducting a null-move search. We make a null move, i.e., we merely swap the side whose turn it is to move. (Note that this cannot be done in positions where that side is in check, since the resulting position would be illegal. Also, two null moves in a row are forbidden, since they result in nothing [13].) We then conduct a regular search with reduced depth and save the returned value. This value can be treated as a lower bound on the position, since the value of the best (legal) move has to be better than that obtained from the null move. If this value is greater than or equal to the current upper bound (i.e., $value \geq \beta$), it results in a cutoff (or what is called a fail-high). Otherwise, if it is greater than the current lower bound α, we define a narrower search window, as the returned value becomes the new lower bound. If the value is smaller than the current lower bound, it does not contribute to the search in any way. The main benefit of null-move pruning is due to the cutoffs, which result from the returned value of null-move search being greater than the current upper bound. Thus, the best way to apply null-move pruning is by conducting a minimal-window null-move search around the current upper bound β, since such a search will require a reduced search effort to determine a cutoff. A typical null-move pruning implementation is given by the pseudo-code of Fig. 1.

There are positions in chess where any move will deteriorate the position, so that not making a move is the best option. These positions are called *zugzwang* positions. While zugzwang positions are rare in the middle game, they are not an exception in endgames, especially endgames in which one or both sides are left with King and Pawns. Null-move pruning will fail badly in zugzwang positions since the basic assumption behind the method does not hold. In fact, the

```
#define R  2 // depth reduction value
int Search (alpha, beta, depth) {
    if (depth <= 0)
        return Evaluate(); // in practice, Quiescence() is called here
    // conduct a null-move search if it is legal and desired
    if (!InCheck() && NullOk()){
        MakeNullMove();
        // null-move search with minimal window around beta
        value = -Search(-beta, -beta + 1, depth - R - 1);
        UndoNullMove();
        if (value >= beta) // cutoff in case of fail-high
            return value;
    }
    // continue regular alphabeta/PVS search
    ...
}
```

Fig. 1. Standard null-move pruning

null-move search's value is an upper bound in such cases. As a result, null-move pruning is avoided in such endgame positions. Here we remark that in the early 1990s Diepeveen suggested a double null-move to handle zugzwang positions. It is an unpublished idea [12].

As previously noted, the major benefit of null-move pruning stems from the depth reduction in the null-move searches. However, these reduced-depth searches are liable to tactical weaknesses due to the *horizon effect* [5]. A horizon effect results whenever the reduced-depth search misses a tactical threat. Such a threat would not have been missed, had we conducted a search without any depth reduction. The greater the depth reduction R, the greater the tactical risk due to the horizon effect. So, the saving resulting from null-move pruning depends on the depth reduction factor, since a shallower search (i.e., a greater R) will result in faster null-move searches and an overall smaller search tree.

In the early days of null-move pruning, most programs used $R = 1$, which ensures the least tactical risk, but offers the least saving in comparison with other R values. Other reduction factors that were experimented with were $R = 2$ and $R = 3$. Research conducted over the years, most extensively by Heinz [20], showed that in his program, DarkThought, $R = 2$ performed better than $R = 1$ and $R = 3$.

Donninger [13] was the first to suggest an adaptive rather than a fixed value for R. Experiments conducted by Heinz in his article on adaptive null-move pruning [20] showed that an adaptive rather than a fixed value can be selected for the reduction factor. By using $R = 3$ in upper parts of the search tree and $R = 2$ in its lower parts (close to the leaves) pruning can be achieved at a smaller costs (as null-move searches will be shallower) while the overall tactical strength will be maintained.

Several methods have been suggested for enabling null-move pruning to deal with zugzwang positions, but mostly at a heavy cost of making the search much more expensive [16,24]. In our 2002 paper, we introduced *verified null-move pruning* [10], which manages to cope with most zugzwang positions, with minimal additional cost. In verified null-move pruning, whenever the shallow null-move search indicates a fail-high, instead of cutting off the search from the current node, the search is continued with reduced depth. Only if another null-move fail-high occurs in the subtree of a fail-high reported node, then a cutoff will take place. Using $R = 3$ in all parts of the search tree, our experimental results showed that the size of the constructed tree was closer to that of standard $R = 3$ rather than $R = 2$ (i.e., considerably smaller tree in comparison to that constructed by using standard $R = 2$), and greater overall tactical accuracy than standard null-move pruning.

So far, all publications regarding null-move pruning considered at most a reduction value of $R = 3$, and any value greater than that was considered far too aggressive for practical use. In the next section we present our *extended null move reductions* algorithm which uses an aggressive reduction value of $R = 4$, by bringing together verified and adaptive principles.

3 Extended Null-Move Reductions

In this section we describe how we combine adaptive and verified null-move pruning concepts into our *extended null move reductions* (NMR), which enable us to use an aggressive reduction value of $R = 4$.

The greater the reduction value R is, the faster will the null-move search be, which will have a large impact on the overall size of the search tree. Thus, using $R = 4$ instead of the common values of $R = 2$ and $R = 3$ would construct a smaller search tree, enabling the program to search more deeply. However, as explained in the previous section, greater R values result in overlooking more tactical combinations. In other words, the benefit of deeper search comes at the cost of taking a greater risk of missing correct moves.

The basic idea behind NMR is using the null-move concept for reducing the search depth only, instead of pruning it altogether. Whenever the null-move search returns a value greater or equal to the upper bound, indicating fail-high ($value \geq \beta$), we reduce the depth and continue the normal search. This concept is different from verified null-move pruning where a fail-high in the subtree of a fail-high reported node results in an immediate cutoff, while in NMR, the subtree is not treated any differently.

There are some similarities between NMR and Feldmann's *fail high reductions* (FHR) [15]. In FHR, in each node a static evaluation is applied, and if the value is greater than or equal to β, the remaining depth is reduced by one ply. The major difference between NMR and FHR is that in the former the decision to reduce the depth is made after a dynamic search, while in the latter the decision is static only. In other words, in subsequent iterations when we revisit the current position, the null-move search will be deeper accordingly,

while the static evaluation at the current position will always return the same value, regardless of the search depth. As we mentioned in the Introduction, null-move pruning succeeded where other forward-pruning methods failed, thanks to basing the pruning decision on dynamic criteria.

Using the null-move concept for depth reduction instead of pruning has the advantage of reducing the tactical weaknesses caused by the horizon effect, since by continuing the search we may be able to detect threats which the shallow null-move search overlooked. Additionally, since NMR does not cutoff based on a fail-high, it is completely impervious to zugzwangs (while verified null-move manages to deal successfully with most zugzwangs, it is not completely impervious since the subtree of the fail-high node is searched normally). Thus, NMR facilitates the usage of the null-move concept even in endgames where zugzwangs are frequent.

Obviously, the disadvantage of NMR is that it has to search a larger tree in comparison to standard null-move pruning with the same R value, as the latter terminates the search at the node immediately upon a fail-high. Considering the pros and cons, the success of NMR depends on the result of this cost benefit analysis. Our experiments in the next section show that the benefit from the reduced searches justifies their additional cost.

So far, we mentioned that whenever the null-move search indicates a fail-high, in NMR we reduce the search depth and continue the normal search. The

```
// depth reduction values for null-move search
#define MAX_R  4
#define MIN_R  3
#define DR     4 // depth reduction value for normal search
int Search (alpha, beta, depth) {
    if (depth <= 0)
        return Evaluate(); // in practice, Quiescence() is called here
    // conduct a null-move search if it is legal and desired
    if (!InCheck() && NullOk()){
        MakeNullMove();
        R = depth > 6 ? MAX_R : MIN_R ;
        // null-move search with minimal window around beta
        value = -Search(-beta, -beta + 1, depth - R - 1);
        UndoNullMove();
        if (value >= beta) { // reduce the depth in case of fail-high
            depth -= DR;
            if (depth <= 0)
                return Evaluate();
        }
    }
    // continue regular alphabeta/PVS search
    ...
}
```

Fig. 2. Extended null-move reductions

success of NMR depends on the depth reduction (DR) applied here. Reducing the remaining depth by only one ply ($DR = 1$) is too conservative, as the remaining search will still be expensive. Our experiments showed that together with a reduction value of $R = 4$ for null-move search, the best reduction value for the remaining search depth is also an aggressive reduction of 4 plies ($DR = 4$). Reducing the remaining depth by a large number reduces the additional cost in comparison to standard $R = 4$ where the search is cutoff immediately.

Finally, to make this aggressive configuration safer, we also incorporate the adaptive null-move concept, i.e., we use a reduction value of $R = 3$ near leaf nodes. Using this adaptive $R = 3 \sim 4$ makes the null-move search less susceptible to overlooking tactics, while keeping the search tree small enough to justify the additional cost. Our results in the next section show that NMR with $R = 3 \sim 4$ and depth reduction of $DR = 4$ outperforms other variations of null-move pruning. Implementation of our extended null-move reductions is very easy in a program already using null-move pruning. Figure 2 shows our NMR implemented around the existing standard null-move pruning code (additions are in bold).

4 Experimental Results

Before discussing the performance of NMR in comparison to other null-move pruning variations, we would like to discuss briefly some basic issues about experimental results in computer chess. Most published papers compare various search methods to each other using fixed depth tests. Usually both method A and method B search the same test suites to fixed depths, and then the results (and number of solved positions) are compared. If method A produces a smaller tree (fewer nodes at the fixed depth) and also solves more positions, then it can be safely concluded that method A outperforms method B. However, in many cases the results will not be so clear. For example, comparing standard null-move pruning with $R = 2$ and $R = 3$, the latter constructs a smaller tree, but solves fewer positions at the fixed depth search.

Fixed time tests, in contrast to fixed depth tests, allow for an objective comparison of various methods. For example, method A can sometimes find the correct move a ply or two later than method B (e.g., because it uses a more aggressive pruning), but considering the elapsed time, method A finds the solution faster. In this case, it would be correct to say that method A performs better, even though in a fixed depth comparison method B solves more positions.

The second issue is which test suites to use. Traditionally, three standard test suites have been used for measuring tactical strength, namely Encyclopedia of Chess Middlegames (ECM), Win at Chess (WAC), and Winning Chess Sacrifices (WCS). While for many years these three test suites posed serious challenges to computer programs, today thanks to the fast hardware most of these positions succumb to the processing power in a fraction of a second. This is natural, as these three test suites were intended for testing humans not machines. Amongst the abovementioned test suites, ECM is the only one which poses some challenge to the engines, provided the time per position is limited to a small value. We

Table 1. Number of ECM positions solved by each engine (time: 5s per position)

Junior 10	Fritz 8	Shredder 10	Hiarcs 9	Crafty 19	Falcon
681	640	639	642	593	644

Table 2. Total node count of standard $R = 1$, 2, 3, and 4 and NMR $R = 3 \sim 4$ for Crafty benchmark

Std $R = 1$	Std $R = 2$	Std $R = 3$	Std $R = 4$	NMR $R = 3 \sim 4$
42,248,908 (+390.9%)	21,554,578 (+150.45%)	11,510,995 (+33.75%)	8,254,261 (-4.09%)	8,606,334 -

Table 3. Number of ECM positions solved by each method (time: 5s per position)

Std $R = 2$	Std $R = 3$	Std $R = 4$	Adpt $R = 2 \sim 3$	NMR $R = 3 \sim 4$
627	635	632	636	644

used the ECM test suite consisting of 879 positions, with 5 seconds per position. To double check the results (and avoid external interferences with CPU time allocations) we ran each test twice, to make sure the same results are obtained.

We conducted our experiments using Falcon, a grandmaster-level chess program which has successfully participated in two World Computer Chess Championships (7th place in 2004 World Computer Chess Championship, and 3rd place in 2004 World Computer Speed Chess Championship). Falcon uses NegaScout/PVS [9,25] search, with enhancements like internal iterative deepening [2,27], dynamic move ordering (history+killer heuristic) [1,17,26], multi-cut pruning [7,8], selective extensions [2,4] (consisting of check, one-reply, mate-threat, recapture, and passed pawn extensions), transposition table [22,29], futility pruning near leaf nodes [19], and blockage detection in endgames [11]. Table 1 compares Falcon's tactical performance to other top tournament-playing engines. The results show that Falcon's tactical strength is on par with the strongest chess programs today.

Before we compare the tactical strength of various methods, we use a fixed depth benchmark of six positions (Crafty benchmark, see Appendix A) to show how significant the impact of the reduction value R is. Table 2 provides the total node count, comparing standard null-move pruning with reduction values of $R =$ 1, 2, 3, and 4, and NMR using $R = 3 \sim 4$ and $DR = 4$. The results clearly show that the R value has a critical role in determining the size of the constructed search tree. The table further shows that as far as node count is concerned, NMR with $R = 3 \sim 4$ is close to standard null-move with $R = 4$. But as discussed above, this table merely shows that the greater the R value is, the deeper the engine will be able to search, saying nothing about the tactical strength.

To compare the overall tactical performance, we let standard $R = 2$, 3 and 4, adaptive $R = 2 \sim 3$ and NMR $R = 3 \sim 4$ process the ECM test suite with 5

Table 4. Number of ECM positions solved by each method (time: 5s per position)

Std $R = 4$	Adpt $R = 3 \sim 4$	NMR $R = 4$	NMR $R = 3 \sim 4$
632	637	640	644

Table 5. Number of ECM positions solved by NMR using various DR values (time: 5s per position)

$DR = 1$	$DR = 2$	$DR = 3$	$DR = 4$
633	641	638	644

Table 6. 1000 self-play matches between two versions of FALCON using NMR $R = 3 \sim 4$ and Adpt $R = 2 \sim 3$, at 10 minutes per game (W% is the winning percentage, and RD is the Elo rating difference)

Match	Result	Score	W%	RD
NMR $R = 3 \sim 4$ vs. Adpt $R = 2 \sim 3$	+309 -217 =474	546.0 - 454.0	54.6%	+32

seconds per position. Table 3 provides the results. These results show that NMR $R = 3 \sim 4$ performs better than the others. We also see that standard $R = 3$ slightly outperforms both standard $R = 2$ and standard $R = 4$, with adaptive $R = 2 \sim 3$ faring about the same.

In order to see what contributes to the success of NMR $R = 3 \sim 4$, we break it down to its components. Table 4 shows a comparison of standard $R = 4$, adaptive $R = 3 \sim 4$, NMR $R = 4$, and NMR $R = 3 \sim 4$. The results show that both adaptive $R = 3 \sim 4$ and NMR $R = 4$ outperform standard $R = 4$, which explains why their combination, NMR $R = 3 \sim 4$, provides the best outcome.

Finally, in all our results above we used a depth reduction value of 4 ($DR = 4$), i.e., whenever a fail-high is indicated, the depth is reduced by 4 plies. Table 5 compares other values for DR. The results show that a value of 4 performs best.

The results so far showed that NMR $R = 3 \sim 4$ solves more positions in comparison to other methods, with adaptive $R = 2 \sim 3$ coming second. To test how NMR fares in practice, we ran 1000 self-play matches between two versions of FALCON, one using NMR $R = 3 \sim 4$ and the other using adaptive $R = 2 \sim 3$, at a time control of 10 minutes per game. Table 6 provides the results.

The results of 1000 self-play matches show that NMR $R = 3 \sim 4$ outperforms adaptive $R = 2 \sim 3$ by about 32 Elo points (see Appendix B for calculation of expected Elo difference for self-play matches). Even though this is a small rating difference, the large number of games (1000) allows for obtaining a high level of statistical confidence. At 95% statistical confidence (2 standard deviations), the rating difference is 32±16 Elo, and at 99.7% statistical confidence (3 standard deviations) the rating difference is 32±24 Elo. That is, NMR $R = 3 \sim 4$ is superior to adaptive $R = 2 \sim 3$ with a statistical confidence of over 99.7%.

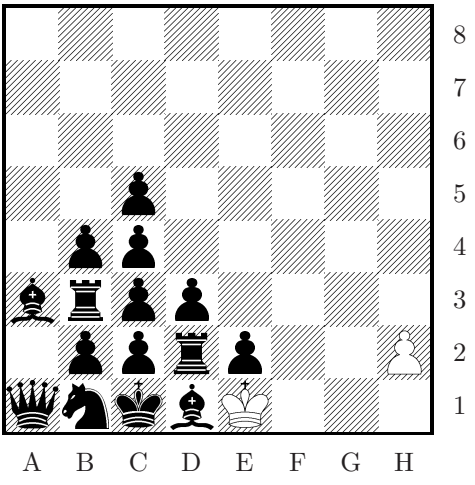


Fig. 3. 1. h3 mates in 15

Table 7. Analysis of the position in Fig. 3. All the engines are given infinite time until they reach their maximum depth.

	Junior 10	Fritz 8	Shredder 10	Hiarcs 9	Crafty 19	Falcon
Move (score)	1.h4 (0.00)	1.h3 (0.00)	1.h3 (#15)	1.h4 (0.00)	1.h4 (0.00)	1.h3 (#15)
Depth	62	60	31	30	60	30

Finally, in late endgames, where zugzwangs are abundant, standard null-move pruning is completely crippled. In contrast, NMR can be safely applied to all stages of the game. The position appearing in Fig. 3, while being a constructed position unlikely to occur in a real game, shows how strong the effect of zugzwang on null-move pruning can be. The only correct move in this position is 1. h3 resulting in mate in 15. The other move 1. h4, results in a draw. Table 7 shows what each engine plays in this position, given infinite time. Falcon and Shredder instantly declare mate in 15 with 1. h3 at the depth of 30 plies (this suggests that Shredder is probably also applying some verification process to null-move pruning). The other engines search to their maximum search depth, all of them declaring a draw. Fritz produces the correct move 1. h3 but with a draw score, suggesting that it has just randomly picked 1. h3 instead of 1. h4.

5 Conclusion

In this article we introduced *extended null-move reductions*, which outperformed conventional null-move pruning techniques both in tactical tests and in long

series of self-play matches. This method facilitates a safe use of the aggressive reduction value of $R = 4$, which is widely considered as too aggressive for practical use. It results in a considerably smaller search tree, enabling the program to search more deeply, thus improving its tactical and positional performance. Moreover, NMR, by the fact that it does not prune based on fail-high, is impervious to zugzwang, and so it can be safely employed in all stages of the game.

NMR and its modified versions have been evolving in FALCON for the past six years, and the results have been promising. In this paper we provided a small fraction of the experiments we have conducted during this period. However, despite our success, we would like to be cautious with any generalization. FALCON has an aggressively tuned king-safety evaluation, and uses many extensions in its search that enable it to spot faster tactical combinations. As such, it is possible that our aggressive method works in FALCON because other components of the engine are tuned for detecting tactics, and they "cover" the blind spots of our NMR.

We believe the main contribution of this paper is that it presents a method for successful incorporation of the seemingly impractical value of $R = 4$ within the null-move search, and even if our method does not achieve exactly the same result in another program, we believe trying other implementations using $R = 4$ is worthy of experimenting with, due to the high potential reward.

In this paper we presented one of the enhancements we have developed during the past few years. It is very probable that our method, or improved incarnations of it, are independently developed by the programmers of other top chess engines.

Acknowledgments. We would like to thank Vincent Diepeveen for his enlightening remarks and suggestions. We would also like to thank the two anonymous referees for their helpful comments.

References

1. Akl, S.G., Newborn, M.M.: The principal continuation and the killer heuristic. In: Proceedings of the 5th Annual ACM Computer Science Conference, pp. 466–473. ACM Press, Seattle (1977)
2. Anantharaman, T.S.: Extension heuristics. ICCA Journal 14(2), 47–65 (1991)
3. Beal, D.F.: Experiments with the null move. In: Beal, D.F. (ed.) Advances in Computer Chess 5, pp. 65–79. Elsevier Science Publishers, Amsterdam (1989)
4. Beal, D.F., Smith, M.C.: Quantification of search extension benefits. ICCA Journal 18(4), 205–218 (1995)
5. Berliner, H.J.: Chess as Problem Solving: The Development of a Tactics Analyzer. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA (1974)
6. Birmingham, J.A., Kent, P.: Tree-searching and tree-pruning techniques. In: Clarke, M.R.B. (ed.) Advances in Computer Chess 1, pp. 89–107. Edinburgh University Press, Edinburgh (1977)
7. Björnsson, Y., Marsland, T.: Multi-cut pruning in alpha-beta search. In: Proceedings of the 1st International Conference on Computers and Games, pp. 15–24 (1998)
8. Björnsson, Y., Marsland, T.: Multi-cut alpha-beta-pruning in game-tree search. Theoretical Computer Science 252(1-2), 177–196 (2001)

9. Campbell, M.S., Marsland, T.A.: A comparison of minimax tree search algorithms. Artificial Intelligence 20(4), 347–367 (1983)
10. David-Tabibi, O., Netanyahu, N.S.: Verified null-move pruning. ICGA Journal 25(3), 153–161 (2002)
11. David-Tabibi, O., Felner, A., Netanyahu, N.S.: Blockage detection in pawn endings. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) CG 2004. LNCS, vol. 3846, pp. 187–201. Springer, Heidelberg (2006)
12. Diepeveen, V.: Private communication (2008)
13. Donninger, C.: Null move and deep search: Selective search heuristics for obtuse chess programs. ICCA Journal 16(3), 137–143 (1993)
14. Feist, M.: The 9th World Computer-Chess Championship: Report on the tournament. ICCA Journal 22(3), 155–164 (1999)
15. Feldmann, R.: Fail high reductions. In: van den Herik, H.J., Uiterwijk, J.W.H.M. (eds.) Advances in Computer Chess 8, pp. 111–128. Universiteit Maastricht (1996)
16. Goetsch, G., Campbell, M.S.: Experiments with the null-move heuristic. In: Marsland, T.A., Schaeffer, J. (eds.) Computers, Chess, and Cognition, pp. 159–168. Springer, New York (1990)
17. Gillogly, J.J.: The technology chess program. Artificial Intelligence 3(1-3), 145–163 (1972)
18. Hammilton, S., Garber, L.: Deep Blue's hardware-software synergy. IEEE Computer 30(10), 29–35 (1997)
19. Heinz, E.A.: Extended futility pruning. ICCA Journal 21(2), 75–83 (1998)
20. Heinz, E.A.: Adaptive null-move pruning. ICCA Journal 22(3), 123–132 (1999)
21. Hsu, F.-h.: IBM's DEEP BLUEchess grandmaster chips. IEEE Micro 19(2), 70–80 (1999)
22. Nelson, H.L.: Hash tables in CRAY BLITZ. ICCA Journal 8(1), 3–13 (1985)
23. Newborn, M.M.: Computer Chess. Academic Press, New York (1975)
24. Plenkner, S.: A null-move technique impervious to zugzwang. ICCA Journal 18(2), 82–84 (1995)
25. Reinefeld, A.: An improvement to the Scout tree-search algorithm. ICCA Journal 6(4), 4–14 (1983)
26. Schaeffer, J.: The history heuristic. ICCA Journal 6(3), 16–19 (1983)
27. Scott, J.J.: A chess-playing program. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 255–265. Edinburgh University Press, Edinburgh (1969)
28. Slagle, J.R.: Artificial Intelligence: The Heuristic Programming Approach. McGraw-Hill, New York (1971)
29. Slate, D.J., Atkin, L.R.: Chess 4.5 – The Northwestern University chess program. In: Frey, P.W. (ed.) Chess Skill in Man and Machine, 2nd edn., pp. 82–118. Springer, New York (1983)

Appendix

A Experimental Setup

Our experimental setup consisted of the following resources:

- 879 positions from *Encyclopedia of Chess Middlegames* (ECM).
- Falcon, Junior 10, Fritz 8, Shredder 10, Hiarcs 9, and Crafty 19 chess engines, running on AMD 3200+ with 1 GB RAM and Windows XP operating system.

- Fritz 8 interface for automatic running of test suites and self-play matches (FALCON was run as a UCI engine).
- CRAFTY benchmark for fixed depth search, consisting of the following six positions:

```
D=11: 3r1k2/4npp1/1ppr3p/p6P/P2PPPP1/1NR5/5K2/2R5 w - - 0 1 D=11:
rnbqkb1r/p3pppp/1p6/2ppP3/3N4/2P5/PPP1QPPP/R1B1KB1R w KQkq - 0 1
D=14: 4b3/p3kp2/6p1/3pP2p/2pP1P2/4K1P1/P3N2P/8 w - - 0 1 D=11:
r3r1k1/ppqb1ppp/8/4p1NQ/8/2P5/PP3PPP/R3R1K1 b - - 0 1 D=12:
2r2rk1/1bqnbpp1/1p1ppn1p/pP6/N1P1P3/P2B1N1P/1B2QPP1/R2R2K1 b - - 0 1
D=11: r1bqk2r/pp2bppp/2p5/3pP3/P2Q1P2/2N1B3/1PP3PP/R4RK1 b kq - 0 1
```

B Elo Rating System

The Elo rating system, developed by Prof. Arpad Elo, is the official system for calculating the relative skill levels of players in chess. Given the rating difference (RD) of two players, the following formula calculates the expected winning rate (W, between 0 and 1) of the player:

$$W = \frac{1}{10^{-RD/400} + 1}$$

Given the winning rate of a player, as is the case in our experiments, the expected rating difference can be derived from the above formula:

$$RD = -400 \log_{10}(\frac{1}{W} - 1)$$

GTQ: A Language and Tool for Game-Tree Analysis

Jónheiður Ísleifsdóttir and Yngvi Björnsson

School of Computer Science, Reykjavík University, Reykjavík, Iceland
{jonheiduri02,yngvi}@ru.is

Abstract. The search engines of high-performance game-playing programs are becoming increasingly complex as more and more enhancements are added. To maintain and enhance such complex engines is a challenging task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. In this paper we introduce the Game Tree Query Language (GTQL), a query language specifically designed for analyzing game trees. The language can express queries about complex game-tree structures, including hierarchical relationships and aggregated attributes over subtree data. We also discuss the design and implementation of the Game Tree Query Tool (GTQT), a software tool that allows efficient execution of GTQL queries on game-tree log files. The tool helps program developers to gain added insight into the search process of their engines, as well as making regression testing easier. Furthermore, we use the tool to analyze and find interesting anomalies in search trees generated by a competitive chess program.

1 Introduction

The development of high-performance game-playing programs for board games is a large undertaking. The search engine and the position evaluator, the two core parts of any such program, become quite sophisticated when all the necessary bells and whistles have been added [1, 7, 8, 18]. To maintain and enhance such complicated software is a challenging task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. A standard software-engineering approach for verifying that new modifications do not break existing code is to use *regression testing*. To a large extent that approach is what game-playing program developers use. They keep around large suites of test positions and verify that a modified program evaluates them correctly and plays the correct move. Additionally, new program versions play a large number of games against different computer opponents to verify that the newly added enhancements result in genuine improvements. Nonetheless, especially when it comes to the search, it can be difficult to detect abnormalities; they may stay hidden for a long time without surfacing. They can be subtle things such as the search extending useless lines too aggressively, or poor move ordering resulting in unnecessarily late cutoffs. Neither of the above abnormalities result in erroneous results, but may instead degrade the efficiency of the search unnecessarily.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 217–228, 2008.

To detect these anomalies one must typically explore and gather statistics about the search process.

In this paper we introduce *Game-Tree Query Language (GTQL)*, a language specifically designed for querying game trees, and expand on previous work [5] by doing a thorough empirical analysis where GTQL queries are used to analyze and look for anomalies in search trees generated by a third-party competitive chess program. As demonstrated, the query language allows the game-program developers to gain better insight into the behavior of the search process of their programs and makes regression testing easier. The developer can now keep around a set of pre-defined queries that check for various wanted or unwanted search behaviors (such as too aggressive extensions or large quiescence searches). When a new program version is tested, it can be instructed to generate log files containing the search trees. The queries are then run against the logs to verify that the search is behaving in accordance with expectations. This has the potential of substantially shorten the testing process as unwanted behaviors can be detected early, as opposed to after playing hundreds of test games or, in the worst case, never.

The paper is organized as follows. In the next section we describe the syntax and semantics of the language, followed by a section giving an overview of the usage and implementation of *Game-Tree Query Tool (GTQT)*, a tool for effectively executing GTQL queries. The efficiency and scalability of the tool is then empirically evaluated, and the tool used to detect anomalies in the search of the chess program Fruit. Finally, we present conclusions and discuss future work.

2 The Game-Tree Query Language

A GTQL query consists of three parts: a *node-expression* part, a *child-expression* part, and a *subtree-expression* part:

```
node:<node-expression>;
child:<child-expression>;
subtree:<subtree-expression>
```

The keywords *node*, *child*, and *subtree* indicate the type of the expression that follows. The query parts must be listed in the order given above, separated by a semi-colon, but any unwanted parts can be omitted.

2.1 Query and Expression Evaluation

To be valid, expressions must be formed such that they evaluate to either true or false. By default a query returns the set of nodes in a game tree that fulfil the query, that is, the nodes for which all query parts evaluate to true. An example query is provided below:

```
node: type = PVNode;
child: count([]type = type) >= 5;
subtree: count(*) > 1000
```

Table 1. Operators listed by precedence

Operator	Type	Arity
[ ], [<]	Hierarchical	unary
&	Attribute	binary
<,>, >=, <=, =, ! =	Relational	binary
not	Logical	unary
and	Logical	binary
or	Logical	binary

The query asks for nodes where the principal variation (PV) of the search is changing frequently (this could, e.g., be an indication of a bad move-ordering mechanism). The node expression evaluates to true only at PV nodes; the child expression counts the number of child nodes that are of the same type as the parent (i.e., also PV nodes) and returns true if there are at least five such child nodes; the subtree expression further limits the set of PV nodes that can fulfill the query by demanding their subtree being of a minimum size — this is done to exclude PV nodes close to the leaves where frequent PV changes may occur naturally.

The language is case sensitive and its expressions consist of *attributes*, *constants*, *operators*, and *functions*. *Attributes* refer to data fields associated with the nodes stored in the game-tree file being queried. For each node several attributes are stored, two of which are always present (*node_id* and *last_move*) while others are optional. The optional attributes are typically algorithm and domain dependent and may contain whatever information the users decide to log in their game-playing programs (e.g., information about the search window passed to a node, the value returned, the type of the node, etc.). In the above example *type* is an attribute telling whether a node is a PV, CUT, or an ALL node. Attribute names follow a naming convention where a name starts with a letter and is then optionally followed by a series of characters consisting of letters, digits, and the underscore character. Also, an attribute name may not be the same as a reserved keyword in the language. *Constants* are either numeric integral types (i.e., integer numbers) or user-defined names (e.g., *PVNode* in our example query). The same naming convention is used for constant names as for attribute names. Information about attribute and constant names available to a query are stored in the game-tree file being queried.

The language operators fall into four categories: *hierarchical*, *attribute*, *relational*, and *logical* operators. They are listed in Table 1 in a decreasing order of precedence. The evaluation of operators of equal precedence is left-to-right associative. The *hierarchical* operators are used as prefixes to attribute names, and identify the hierarchical relationship of the referenced node in relation to the current node (the one being evaluated in the node expression). Currently, there are two such operators defined, and they may be used only in child expressions. The first operator, [ ], is prefix referring to the child node currently evaluated. In our example, the child expression has such an operator for comparing the type of the child nodes to the type of the node evaluated by the node expression (the

parent). The second hierarchical operator [<], not shown in the example, stands for the previously evaluated child. It can be used to compare two consecutive child nodes (e.g., to see if a node is being examined). The *attribute* operator, &, is essentially an inclusive bitwise *and*, and is used to extract flag bits from attribute fields. For example, a single node may be flagged simultaneously as a *quiescence* node and as belonging to a *null-move* search. The *relational* operators test for equality or inequality of attributes, constants, function results, and numbers, and the *logical* operators allow one to form arbitrarily complex expressions by combining Boolean expressions. Parentheses can be used to control precedence and order of evaluation.

There is only one function in the language, the `count(sub-expression)` function, and it returns the number of nodes in the expression scope (i.e., tree, children, or subtree) that evaluate to true. Functions cannot be used recursively, that is, the expression inside *count* cannot contain a call to *count*. The wild-card character * may be used within the function instead of an expression to refer to the empty expression, which always evaluates to true. Note that because expressions must evaluate to either true or false, the count function must be used with a relational operator, e.g. `count(*)>0`. The only exception is when the function is used stand-alone in a node expression. In that case, the query returns the actual count as opposed to a set of nodes. This is useful for gathering statistics about the tree, e.g., as in the example below where the total number of PV nodes in the tree is being counted:

```
node: count(type = PVNode)
```

More query examples are provided later in the paper. However, for a more thorough explanation of the syntax and semantics of GTQL, as well as for additional query examples, we refer interested readers to [5, 14].

3 Game-Tree Query Tool

The *Game-Tree Query Tool (GTQT)* is a software for parsing and executing GTQL queries. It is a console application that runs from a command line. It is implemented in C++ and runs on both Linux and Windows (as well as other platforms that support ANSI compliant C++ compilers).

Below we give a brief overview of the one-pass algorithm used for executing the queries. The algorithm is capable of answering any single query, no matter how complex, in a single traversal of the game tree. The input to the program is a set of queries and a game-tree log file. In addition to the game-tree data (the attribute values of the nodes) the file stores meta-data, such as the names of the attributes and constants available to the query and information about the layout of the file. The tool, after processing and validating the meta-data, parses and syntactically checks the queries before executing them. For a more detailed discussion of the query execution algorithm, the logging mechanism, and the usage of the GTQL tool we refer readers to [14].

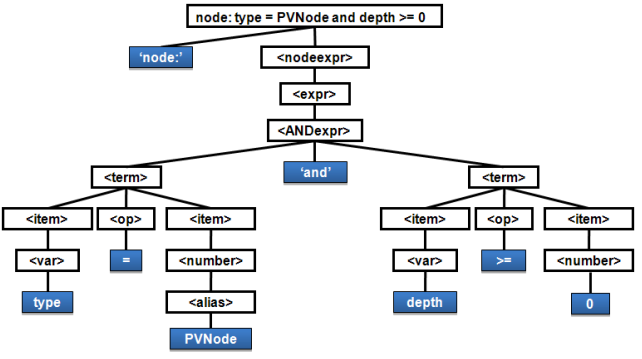


Fig. 1. An example parse tree

3.1 Parsing a Query

Queries are parsed using a recursive-decent parser. A separate parse tree is built for each query. An example parse tree is shown in Fig. 1, along with the query it represents. A parse tree consists of several different types of parse nodes, depending on the type of operator (e.g., relational or logical), term, or expression being evaluated. Most parse nodes return a Boolean value when evaluated, representing whether the corresponding expression evaluated to true or false for any given node in the game tree. Typically, the result of an evaluation on a game-tree node depends on the attribute values stored with the node. For example, in Fig. 1 the values of both the *type* and *depth* fields are required for evaluating the query; for nodes where *type* is equal to *PVNode* and *depth* is greater or equal to zero the query evaluates to true, but to false for all other nodes.

A special provision must be taken for queries containing the function *count*, as it returns a count based on data accumulated over many nodes. Such queries cannot be evaluated until after all data nodes in the expression scope have been traversed. In that case, in addition to the attribute values, a special structure containing count information accumulated over the scope (e.g., a subtree) of the query must be provided. This structure is called a *counter*. Node-expressions can only contain one count function, whereas both subtree- and child-expression can contain many such functions; for such expressions a list of counter structures is required. Note that the counter lists and counter structures are not stored as a part of the parse tree because our query execution algorithm may have to execute several counter based *query instances* concurrently. Instead multiple instances of the query are created, each using its own set of counters (see later).

3.2 Executing a Query

Although the tool is used for post-processing game-tree logs, time is still of some essence when evaluating large game trees. The query execution algorithm makes only one pass through the game tree, during which it collects all information needed to answer the query. The algorithm is presented as Algorithm 1. It first

Algorithm 1. DFT-QUERY-EVAL(node)

```
 1: queryInst = null
 2: if nodeExpr.evaluate(node) then
 3:    queryInst = new QueryInstance(subtreeExpr, childExpr)
 4:    queryInstStack.push(queryInst)
 5: children = node.getChildren()
 6: prev = null
 7: for all child in children do
 8:    DFT-QUERY-EVAL(child)
 9:    evalCounterExprs(queryInst, node, child, prev)
10:    prev = child
11: if not queryInstStack.empty() then
12:    if queryInst == queryInstStack.top() then
13:       if subtreeExpr.evaluate(queryInst) and childExpr.evaluate(queryInst) then
14:          addToResult(node)
15:       queryInstStack.pop()
16:       delete queryInst
17: evalCounterExprs(queryInstStack, node)
```

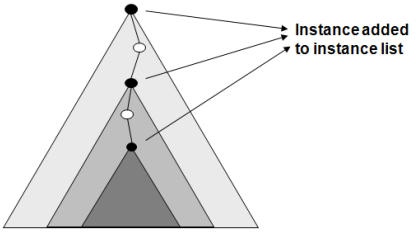


Fig. 2. Subtree scope of different nodes in the same line

checks if the node expression evaluates to true (line 2), and if so a new query instance is created and put onto a so-called query instance stack (line 4). The stack keeps track of active query instances. A single query may have several query instances active at the same time. More specifically, during the recursive depth-first traversal (line 8) of the game tree, each node on the path from the root to the current node where the node expression evaluates to true adds a new query instance. The need for having many query instances open at the same time is because separate counters are required for evaluating the child- and subtree-expressions of each instance, as their subtree scopes differ as shown in Fig. 2. Child- and subtree-expressions can be evaluated (line 13) only when the search backtracks after their corresponding tree scope has been fully traversed. If both expressions evaluate to true, the node is added to set of results (line 14) and the query instance then popped off the stack (line 15). The subroutine *evalCounter-Exprs* updates the counters associated with the `count(<expr>)` expressions. It is called for both child expressions (line 9) and subtree expressions (line 17). In the latter case, counters in all query instances on the stack are updated.

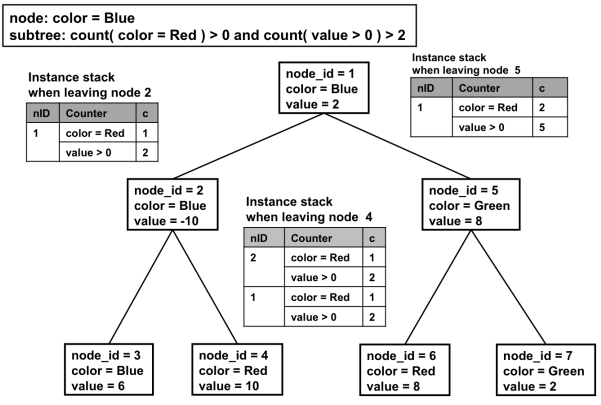


Fig. 3. Example of one-pass query evaluation

An example of the one-pass query-evaluation process is given in Fig. 3. The node-expression part of the query looks for nodes with the color blue. In this example we refer to the nodes by their *node_id*. The node with *node_id*=`1` becomes Node_1. The root is blue so a new query instance is created on the stack. This instance contains two counters: one for the sub-expression `color=Red` and one for `value>0`. The counter stores a pointer to the parse tree of the count sub-expression, and a counter variable initialized to zero (the *c* field in the figure). The traversal continues down the left branch, and because the node-expression is also true for Node_2, an instance is created on the stack for that node as well. An instance is also added for Node_3. Now, because a leaf has been reached, the DFT-QUERY-EVAL algorithm evaluates the subtree-expression for Node_3 based on the instance (the evaluation is false in this case) and backtracks. However, before backtracking the remaining query instances on the stack are updated according to evaluation of Node_3 (the counter for `value>0` is increased by one for both instances). The instances for Node_1 and Node_2 have now been updated and the algorithm has backtracked to Node_2. From there it continues to traverse the children and explores Node_4. This process continues until the entire tree has been traversed. A snapshot of the query instance stack is shown in the figure at selected points (text above the stacks in the figure). The rightmost snapshot shows the stack when the algorithm backtracks back to Node_1 for the last time. We can see that the instance for Node_1 is the only one left on the stack and its counters have been updated several times. Node_1 is now evaluated based on the query instance, the subtree-expression is true, so the node is added to result.

4 Experiments

To demonstrate the potentials of GTQL we used it to analyze game-tree logs generated by a competitive chess program. In this section we report our findings.

Table 2. Game trees used in the experiments

LCT II Position	SD	SSD	Number of nodes
15	8	15	205,199
16	2	19	2,383
17	6	18	197,803
18	10	30	1,671,866
19	5	25	78,165
20	8	24	580,158
21	9	45	2,821,292
22	9	41	5,135,007
23	9	35	1,009,011

4.1 Experimental Setup

For our experiments we used the chess program FRUIT [15], developed by Fabien Letouzey. It was first released in March 2004, and subsequently made a strong appearance in the 2005 World Computer Chess Championship held in Reykjavík [6]. We used version 2.1 of the program, which is the strongest open-source chess engine available (subsequent versions of the program were not open source). The only modification we made to the program was to augment its search engine with code for collecting attribute values and with calls to the game-tree log library.

The chess program was instructed to search nine tactical chess positions taken from the LCT II test suite (positions number 15-23) [16]. This suite is one of several frequently used standard test suites to measure chess programs' performance. On each of the problems the chess program was run until the correct solution was found. For each position, a separate game-tree log was generated for each search iteration. The solution (best move played) was found on iterations varying from the second to the tenth ply, as shown in Table 2. The first column indicates the position within the suite; the second column, SD, shows the search depth of the iteration where the best move was first returned; the third column, SSD, is the maximum search depth reached in that iteration; and the final column is the number of nodes searched in that iteration. In our experiments we used for each position the game-tree log from the iteration where the solution was first found (SD). The experiments were run on a 3GHz Linux-based computer with 2GB of main memory.

4.2 Processing Throughput

We start by measuring the throughput of the query tool. It can process around 500 to 600 thousand nodes per second from the game-tree log, depending on the complexity of the query. The average time complexity of our one-pass query algorithm is $O(n * log(n))$ where n is the number of nodes, so the throughput degrades only slightly with increasingly larger trees [14]. The throughput speed is in the ballpark of how fast chess programs search and log the game trees.

Table 3. Ratio of node types in the game trees

Tree	Num. of nodes	PV nodes	CUT nodes	ALL nodes
Pos_{15}	205,199	0.44%	69.71%	29.86%
Pos_{16}	2,382	14.60%	63.58%	21.82%
Pos_{17}	197,803	0.03%	74.25%	25.73%
Pos_{18}	1,671,866	0.06%	75.87%	24.07%
Pos_{19}	78,165	0.52%	73.32%	26.16%
Pos_{20}	580,158	0.08%	73.19%	26.72%
Pos_{21}	2,821,292	0.10%	72.10%	27.79%
Pos_{22}	5,135,007	0.12%	76.93%	22.95%
Pos_{23}	1,009,011	0.15%	67.52%	32.33%

4.3 Node Type Statistics

Next we asked queries for collecting statistics about the game trees, more specifically the ratio of PV, CUT, and ALL nodes. The queries are shown below:

```
node: count(type = PVNode)
node: count(type = CUTNode)
node: count(type = ALLNode)
```

and the result in Table 3. The result looks as one would expect: very low ratio of PV nodes, and 2 to 3 times more CUT than ALL nodes. The only deviation from this is in Pos_{16} where there is a unusually high ratio of PV nodes, but that is not much of a concern because of the fact that the tree is small (PV changes are not too uncommon in shallow trees). This example, although not providing much additional insight, is good for a sanity check to confirm the expected behavior. The statistics were provided as a demonstration of the type of statistics that can be collected. One must also be a little cautious when working with accumulated statistics, as they may overlook individual anomalies. We thus look more carefully at PV changes in the next subsection.

4.4 Principal Variation Changes

The query below was executed for different values on n:

```
node: type = PVNode;
child: count([]type = type) >= n
```

The result is shown in Table 4. As can be seen, frequent PV changes are uncommon, although there are a few problematic nodes in Pos_{22} that might warrant a further investigation (there are 5 positions with 9 or more PV child nodes).

4.5 Large Quiescence-Search Trees

Quiescence searches are essential in chess programs for evaluating unstable positions — such searches typically include selected captures and even checks. However, because of the frequency of quiescence searches, it is important to

Table 4. Number of PV nodes in each tree with several PV node children

#pv-children	Pos_{15}	Pos_{16}	Pos_{17}	Pos_{18}	Pos_{19}	Pos_{20}	Pos_{21}	Pos_{22}	Pos_{23}
2	91	29	1	87	32	26	243	556	160
5	7	3	0	3	3	2	15	59	20
7	0	0	0	0	0	0	0	6	6
9	0	0	0	0	0	0	0	5	0

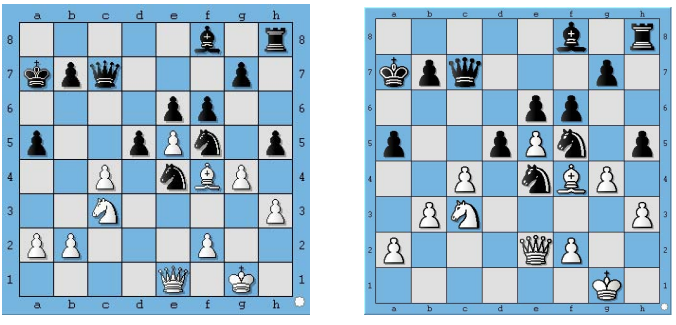


Fig. 4. Chess positions with a large quiescence search

keep their size under control, a process that takes a careful tuning. To obtain more insight into the size of the quiescence searches of Fruit we executed the following query for different values of n:

```
node: flags & QRootNode;
subtree: count(*) > n
```

The attribute *flags* is used in the chess program to mark various node properties, such as whether a node belongs to a research, a null-move search, or a quiescence search. The root of a quiescence search tree is marked as *QRootNode*.

For all positions except one, the quiescence searches were all under 200 nodes (and most much smaller). In Pos_{18} (from the game Vanka - Jansa, Prag 1957), however, four of the quiescence-search subtrees had more nodes. For example, from the two positions shown in Fig. 4 the generated quiescence-search trees were of size 840 and 486 nodes, respectively. This is quite excessive compared to a normal search, and should raise a flag as something that warrants further investigation. This is a good example of how the tool can be used to help identify problems with the search performance.

5 Related Work

To the best of our knowledge GTQL is the first query language specifically designed for game trees. However, several query languages for tree structures exists, including XPath [10] for querying XML data. The navigational abilities of XPath have been used in subsequent languages either by directly supporting

XPath like XQuery [9] does or extending its syntax like is done in LPath [2]. XSQuirrel [17] is a related language for making sub-documents out of existing XML documents. None of the aforementioned languages are well suited for our purpose and do for example not allow aggregation. However, there does exist a chess-specific query language, Chess Query Language (CQL) [11], but it is designed for matching chess positions, not tree structures.

On a further account, there do exist some tools that can be helpful in visualizing game trees. Rémi Coulom presented a visualization technique for search trees using treemaps [12]. Treemaps are based on the idea of taking a rectangle and dividing it into sub-rectangles for each subtree. The first rectangle is split vertically, one rectangle per child. Those rectangles are then split horizontally for each of their children and so on. Although such a technique can give some insight into where the search mainly spends it effort, it is insufficient for detecting most search abnormalities. There do also exist browsers that allow one to navigate through game-tree logs and look at resulting game positions [3, 4, 13].

6 Conclusions

From the above results we may fairly conclude that the new query language and software can aid researchers and game-playing program developers in verifying the correctness of their game-tree search algorithms. The syntax and semantics of the language are explained in such terms that GTQL can be used by others. The GTQ tool expresses queries about complex game-tree structures, including hierarchical relationships and aggregated attributes over subtree data. Last but not least, in this paper we demonstrated the usefulness of the GTQ tool by analyzing and finding abnormalities in the search trees of the competitive chess program Fruit. These are just a few examples of the usefulness of GTQ. The tool is quite flexible as the users decide which information about the game trees to log. For example, by logging static node evaluations one can envision the tool being useful to researchers working on search, e.g., for finding pathological behaviors or for measuring diminishing returns of deeper search.

GTQL is the first query language specifically designed for game trees. There are still many additions and improvements that could be made in future versions of both GTQL and GTQT. For example, the expressiveness of the language could be enhanced, e.g., to include parent relations (and more generally ancestor relations), as well as an extended sibling relation. Also, other functions like *min* and *max* would be useful. Moreover, there are improvements to be made to the implementation; the two most prominent ones are: (1) allowing many queries to be answered simultaneously, and (2) introducing run-time compression/decompression to the game-tree log files as they can quickly grow large. As of now, the tool cannot handle game trees built in parallel. This limitation is worthwhile to be addressed in future versions as multi-core processors are becoming mainstream.

Finally, it is our hope that this work will aid researchers in the field of search algorithms with the tedious process of debugging and verifying the correctness

of their programs, thus saving them countless hours of frustration and grief. The Game-Tree Query Tool is available for download at http://cadia.ru.is.

Acknowledgments. This research was supported by grants from The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract MIRG-CT-2005-017284. We also thank the anonymous referees for their insightful comments.

References

1. Billings, D., Björnsson, Y.: Search and knowledge in Lines of Action. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) ACG. IFIP, vol. 263, pp. 231–248. Kluwer, Dordrecht (2003)
2. Bird, S., Chen, Y., Davidson, S.B., Lee, H., Zheng, Y.: Extending XPath to support linguistic queries. In: Proceedings of Programming Language Technologies for XML (PLANX), Long Beach, California, January 2005, pp. 35–46. ACM Press, New York (2005)
3. Björnsson, Y.: Selective Depth-First Game-Tree Search. PhD thesis, University of Alberta, Canada (June 2002)
4. Björnsson, Y., Ísleifsdóttir, J.: Tools for debugging large game trees. In: Proceedings of The Eleventh Game Programming Workshop, Hakone, Japan (2006)
5. Björnsson, Y., Ísleifsdóttir, J.: GTQL: A query language for game trees. In: Proceedings of The Twelfth Game Programming Workshop, Amsterdam, The Netherlands, pp. 205–216 (2007)
6. Björnsson, Y., van den Herik, H.J.: The 13th world computer-chess championship. ICGA Journal 28(3), 162–175 (2005)
7. Buro, M.: How machines have learned to play Othello. IEEE Intelligent Systems 14(6), 12–14 (1999)
8. Campbell, M., Hoane Jr., A.J., Hsu, F.-h.: Deep blue. Artificial Intelligence 134(1-2), 57–83 (2002)
9. Chamberlin, D.: XQuery: An XML query language. IBM Systems Journal 41(4), 597–615 (2002)
10. Clark, J., DeRose, S.: XML path language (XPath) 1.0. Technical report, W3C Recommendation (1999)
11. Costeff, G.: The Chess Query Language: CQL. ICGA Journal 27(4), 217–225 (2004)
12. Coulom, R.: Treemaps for search-tree visualization. In: Uiterwijk, J.W.H.M. (ed.) The 7th Computer Olympiad Computer-Games Workshop Proceedings (2002)
13. Fortuna, A.: Internet Resource, CHANT: A Tool to View Chess Game Trees (2003), http://chessvortex.com/chant
14. Ísleifsdóttir, J.: GTQL: A Game-Tree Query Language. Master's thesis, Reykjavik University, Iceland (January 2008), http://www.ru.is/?PageID=7094
15. Letouzey, F.: Internet Resource, Fruit Chess (2005), http://www.fruitchess.com
16. Louguet, F.: La Puce Échiquéenne. Internet Resource, LCT II v. 1.21 (2007), http://perso.orange.fr/lefouduroi/testlct2.htm
17. Sahuguet, A., Alexe, B.: Sub-document queries over XML with XSQirrel. In: WWW 2005: Proceedings of the 14th international conference on World Wide Web, pp. 268–277. ACM Press, New York (2005)
18. Schaeffer, J.: One Jump Ahead: Challenging Human Supremacy in Checkers. Springer, Heidelberg (1997)

Probing the 4-3-2 Edge Template in Hex

Philip Henderson and Ryan B. Hayward

Department of Computing Science, University of Alberta,
Edmonton, AB, Canada, T6G 2E8
{ph,hayward}@cs.ualberta.ca
www.cs.ualberta.ca/∼{ph,hayward}

Abstract. For the game of Hex, we find conditions under which moves into a 4-3-2 edge template are provably inferior.

1 Introduction

Hex, the two-player board game invented independently by Piet Hein [8] and John Nash [4,11,12] in the 1940s, is played on a four-sided grid of hexagonal cells. In alternating turns, each player colors an uncolored, or empty, cell with her color (or, if each player has a set of colored stones, by placing a stone of her color on an empty cell). A player wins by connecting her two sides via a set of cells that have her[1] color, as shown in Fig. 1. For more on Hex, see Ryan Hayward and Jack van Rijswijck's paper, Thomas Maarup's webpage, Jack van Rijswijck's webpage, or Cameron Browne's book [3,6,10,13].

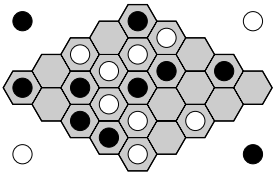


Fig. 1. A Hex board state with a winning White connection

Given a player P (in this paper, B for Black or W for White) and an empty cell c of a board state S, $S + P[c]$ denotes the board state obtained from S by P-coloring c, namely, by coloring c with P's color. See Fig. 2. We denote the opponent of P by $\overline{P}$.

A *game state* $P(S)$ specifies a board position S and the player to move P. With respect to a player P and game states $Q(S_1)$ and $Q(S_2)$, where Q can be P or $\overline{P}$, we write $Q(S_1) \geq_P Q(S_2)$ if $Q(S_1)$ is at least as good for P as $Q(S_2)$ in the following sense: P has a winning strategy for $Q(S_1)$ if P has a winning strategy for $Q(S_2)$. In this case, we say that $Q(S_1)$ P-*dominates* $Q(S_2)$.

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

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 229–240, 2008.

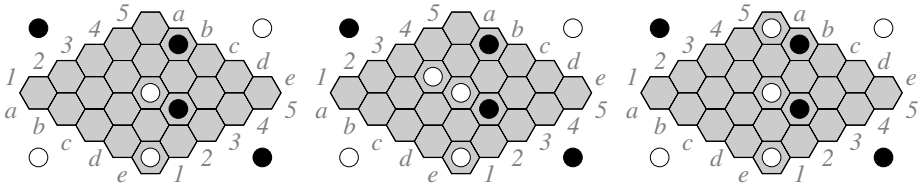


Fig. 2. State S (left), state $T = S + W[b3]$, and state $U = S + W[a5]$

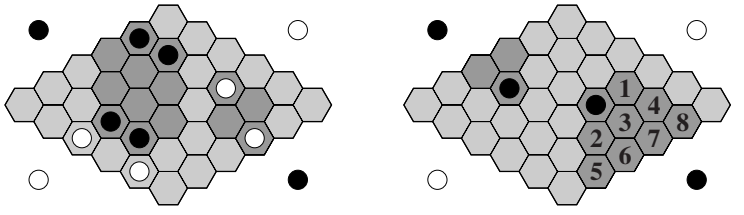


Fig. 3. Two virtual connections (left). A Black edge bridge and 4-3-2 (right).

Consider for example states T and U in Fig. 2. As the reader can check, Black has a winning move in T but no winning move in U, so $B(T) \geq_B B(U)$. Draws are not possible in Hex, so White has a second-player winning strategy for U but not for T, so $B(U) \geq_W B(T)$.

We extend this terminology as follows: with respect to a player P and board states S_1 and S_2, we write $S_1 \geq_P S_2$ if $P(S_1) \geq_P P(S_2)$ and $\overline{P}(S_1) \geq_P \overline{P}(S_2)$. In this case, we say that S_1 *P-dominates* S_2.

With respect to a game state $P(S)$, an empty cell c_1 is *P-inferior* to an empty cell c_2 if c_2 is a P-winning move or c_1 is a P-losing move (equivalently, $\overline{P}(S + P[c_2]) \geq_P \overline{P}(S + P[c_1])$). In this case, we say that c_2 *P-dominates* c_1. Note that domination of game states, board states, and cells is reflexive and transitive.

For example, let S be as shown in Fig. 2 with White to move. In S, b3 loses for White since Black has a winning move in T, and a5 wins for White since Black has no winning move in U. Thus for S, b3 is White-inferior to a5 (equivalently, a5 White-dominates b3).

We write $S \equiv_P T$ if $S \geq_P T$ and $T \geq_P S$. Draws are not possible in Hex, so $S \equiv_P T$ if and only if $S \equiv_{\overline{P}} T$, so we write $\equiv$ in place of $\equiv_P$.

In the search for a winning move, an inferior cell can be pruned from consideration as long as some cell that dominates it is considered. With respect to a board state and a player P, a subset V of the set of empty cells U is *P-inferior* if each cell in V is P-inferior to some cell of $U - V$.

With respect to a board state and a player P, a *virtual connection* is a subgame in which P has a second-player strategy to connect a specified pair of cell sets; thus P can connect the two sets even if $\overline{P}$ has the first move. We say that the cell sets are *virtually connected*, and refer to the empty cells of the virtual connection

as its *carrier*. The left diagram in Fig. 3 shows two virtual connections. The smaller virtual connection, with a two-cell carrier, is often called a *bridge*.

A virtual connection between a set of P-colored cells and one of P's board edges is an *edge template* for P. Two examples are the *edge bridge* and the *edge 4-3-2*, shown in Fig. 3. For more templates, see David King's webpage [9].

Throughout this paper, we refer to an edge 4-3-2 simply as a *4-3-2*, and we refer to a 4-3-2's eight carrier cells by the labels used in Fig. 3. Note that a 4-3-2 is indeed a virtual connection: if White plays at any of {2,5,6}, Black can reply at 4; if White plays at any of {1,3,4,7,8}, Black can reply at 2. The reader can check that a 4-3-2's carrier is minimal: if any of the eight cells belongs to the opponent, the player no longer has a virtual connection.

With respect to a particular virtual connection of a player, a *probe* is a move by the opponent to a carrier cell; all other opponent moves are *external*. In this paper, we explore the question: when are probes of a Black 4-3-2 inferior?

2 Dead, Vulnerable, Captured, and Capture-Dominated

For a board state and a player P, a set of empty cells C is a *P-connector* if P-coloring its cells yields a winning connection; the set is *minimal* if no proper subset is a P-connector. An empty cell is *dead* if it is not on any minimal P-connector. See Fig. 4.

Note that each dead cell is Q-inferior to all other empty cells for both players Q; also, coloring a dead cell an arbitrary color does not change a game state's win/loss value. An empty cell is *P-vulnerable* if some $\overline{P}$-move makes it dead; the cell of this move is a *killer* of the vulnerable cell. Thus, in the search for a P-winning move, dead and P-vulnerable cells can be pruned from consideration.

A set of cells C is *P-captured* if P has a second-player strategy that makes each cell in the set dead or P's color. Since the color of dead cells does not matter, C can be P-colored without changing the value of the board position. For example, the carrier of a Black edge bridge is Black-captured since, for each of the two carrier cells, the cell can be killed by a Black reply at the other carrier cell [5]. An empty cell is *P-capture-dominated*[2] by another empty cell if playing the latter P-captures the former.

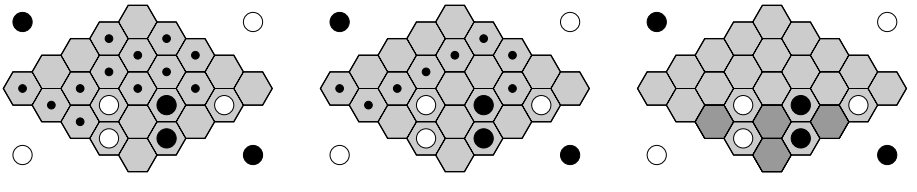


Fig. 4. A Black-connector, a minimal Black-connector, and dead cells

[2] Previous papers on dead cell analysis refer to this simply as *domination* [2,7]. In this paper, we use the term domination in a more general sense.

Note that *vulnerable*, *captured*, and *dominated* are defined with respect to a player; by contrast, *dead* is not. See *Hex and Combinatorics* [6] or *Dead Cell Analysis in Hex and the Shannon Game* [2] for more on inferior cell analysis.

3 A Conjecture

As noted previously, the carrier cells of a Black edge bridge are Black-captured, and so White-inferior to all empty cells. For a 4-3-2, things are not so simple.

As shown in Fig. 5, probes 1,2,4 can each be the unique winning move. Also, as shown in Fig. 6, probes 3,5 can win when probes 1,2,4 do not; however, in the example shown there is also a winning external move, and probes 3,5 merely delay an eventual external winning move. We know of no game state in which one of the probes 3,5,6,7,8 is the unique winning move, nor of a game state in which one of the probes 6,7,8 wins but probes 1,2,4 all lose. Probes 1,2,4 seem generally to be stronger than the others, so we conjecture the following:

Conjecture 1. Probes 3,5,6,7,8 of a Black 4-3-2 are White-inferior.

Thus, for a player P and a particular $\overline{P}$-4-3-2, we conjecture that if P has a winning move, then there is some P-winning move that is not one of the five probes 3,5,6,7,8. In the rest of this paper we find conditions under which the conjecture holds.

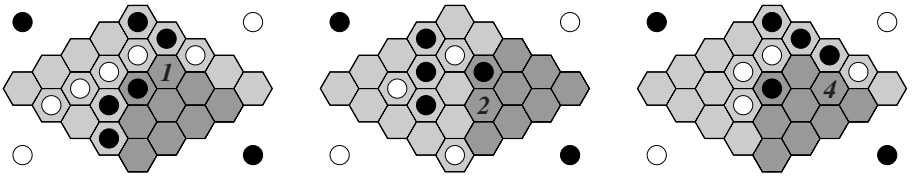


Fig. 5. Only White winning moves: probe 1, probe 2, probe 4

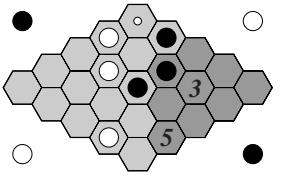


Fig. 6. Only White winning moves: probe 3, probe 5, or the dotted cell

4 Black Maintains the 4-3-2

In Hex, maintaining a particular 4-3-2 is often critical; in such cases, if the opponent ever probes that 4-3-2, the player immediately replies by restoring the

virtual connection. Under these conditions, described in the following theorem, our conjecture holds (except possibly for probe 5, whose status we do not know).

Theorem 1. *Consider a game state with a Black 4-3-2 and White to move. Assume that Black responds to a White probe of this 4-3-2 by restoring the virtual connection. Then each White probe in* $\{3, 6, 7, 8\}$ *is White-inferior.*

To prove the theorem, we will show that it is better for White to probe in $\{1, 2, 4\}$ than in $\{3, 6, 7, 8\}$. To begin, consider possible Black responses to White probes 1,2,4. Against White 1, every other carrier cell maintains the virtual connection; however, Black 2 captures {3,5,6,7}, so {3,5,6,7} are Black-dominated by 2 and need not be considered as Black responses; similarly, Black 4 captures {7,8}. Thus, we may assume: after White 1, Black replies at one of {2,4}; after White 2, Black replies at one of {3,4}; after White 4, Black replies at 2.

We shall show that if White probes at any of {3,6,7,8}, then Black has a response that maintains the 4-3-2 and results in a state where at least one of the following holds: the state is Black-dominated by both states that result after White probes at 1 and Black replies in {2,4}; the state is Black-dominated by both states that result after White probes at 2 and Black replies in {3,4}; the state is Black-dominated by the state that results after White probes at 4 and Black replies at 2.

Our proof of Theorem 1 uses three kinds of arguments. The first two deal with particular forms of domination, which we call path-domination and neighborhood-domination. The third deals directly with strategies. Before presenting the proof, we give some definitions and lemmas.

For a player P and a board state with empty cells c_1 and c_2, we say that c_2 *path-dominates* c_1 if every minimal P-connector that contains c_1 also contains c_2. As the following lemma shows, path-domination implies domination.

Lemma 1. *For a player* P *and empty cells* c_1, c_2 *of a board state* S*, assume that* c_2 *path-dominates* c_1*. Then* $S + P[c_2] \geq_P S + P[c_1]$.

Proof. A P-state is a state in which it is P's turn to move. We prove that $S + P[c_2]$ is P-winning whenever $S + P[c_1]$ is P-winning. Thus, assume P has a winning strategy tree T_1 for $S + P[c_1]$. By definition, T_1 considers all possible $\overline{P}$-continuations for all $\overline{P}$-states and specifies a unique P-winning response in each P-state. Without loss of generality, assume that T_1 continues play until the board is completely filled, namely, it does not stop when a winning path is formed. Thus, all leaves in T_1 appear at the same depth and contain a P-winning path.

Construct a strategy tree T_2 by replacing each occurrence of c_2 in T_1 with c_1. We claim that T_2 is a P-winning strategy tree for $S + P[c_2]$.

First, note that in T_2 the board is played until filled, and that all legal moves for $\overline{P}$ are considered at each stage. Furthermore, a unique P-response is given in each P-state. Thus, T_2 is a valid strategy tree. It remains only to show that each leaf of T_2 has a P-connector.

By contradiction, assume that some leaf L_2 in T_2 has no P-connector. Consider the corresponding leaf L_1 in T_1, attained via the same sequence of moves with

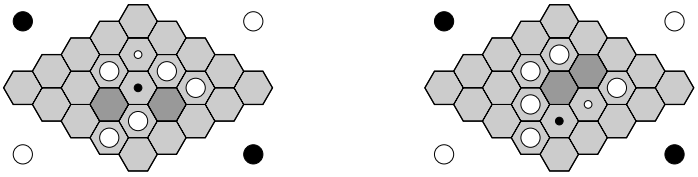


Fig. 7. Killing Black-vulnerable cells without path-domination. The White-dotted cell kills the Black-dotted cell because of White-captured cells that include the shaded cells.

c_1 replaced by c_2. Since L_1 has a P-connector, this connector must use cell c_1, as it is the only cell that can be claimed by P in L_1 and not claimed by P in L_2. However, c_1 is claimed by $\overline{P}$ in L_2, so c_2 is claimed by $\overline{P}$ in L_1. This is a contradiction, as our P-connector in L_1 requires c_2 as well as c_1. Thus, each leaf in T_2 is P-winning. □

If c_2 P-path-dominates c_1, then c_1 is P-vulnerable to c_2. As Fig. 7 shows, the converse does not always hold; it may be that cells captured by the killer are needed to block all minimal connectors.

Lemma 1 yields the following corollary.

Corollary 1. *Let S be a Hex state with empty cells c_1, c_2 such that c_2 P-path-dominates c_1, and c_1 P-path-dominates c_2. Then $S + P[c_1] \equiv S + P[c_2]$.*

Proof. By Lemma 1, $S + P[c_1] \geq_P S + P[c_2]$ and $S + P[c_2] \geq_P S + P[c_1]$. □

Using Lemma 1 and Corollary 1, as well as capturing cells near the edge, we can determine many domination and equivalence relationships between states obtained via the exchange of two moves within the 4-3-2 carrier. We summarize these relationships in Fig. 8, and present two of their proofs as Lemmas 2 and 3. The omitted proofs are similar.

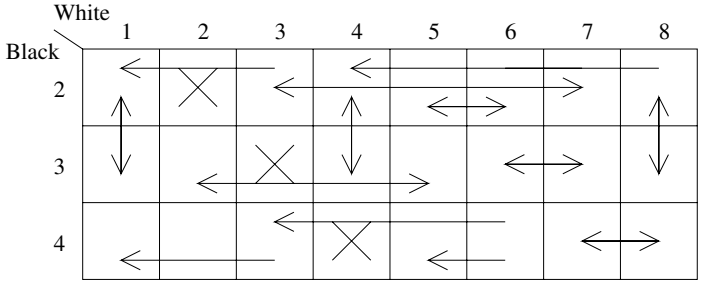


Fig. 8. Some White-domination relations among exchange states. Each arc points from a state to a White-dominating state. Bi-directional arcs indicate equivalent states. X indicates an impossible exchange state. Arcs which follow by transitivity are not shown.

Lemma 2. $S + W[2] + B[3] \equiv S + W[5] + B[3]$.

Proof. $B[3]$ forms an edge bridge, so cells 6 and 7 can be filled-in for Black without changing the value of $S + B[3]$. It can then be seen that all minimal White-connectors that use cell 2 require cell 5, and vice-versa. Thus the result follows from Corollary 1. □

Lemma 3. $S + W[1] + B[4] \geq_W S + W[3] + B[4]$.

Proof. $B[4]$ forms an edge bridge, so cells 7 and 8 can be filled-in for Black without changing the value of $S + B[4]$. It can then be seen that all minimal White-connectors that use cell 3 require cell 1. Now use Lemma 1. □

The P-neighborhood of a cell is the set of all neighbors that are empty or P-colored. A cell c_1 P-dominates a cell c_2 when c_1's P-neighborhood contains c_2's P-neighborhood; in this case, we say that c_1 *P-neighbor-dominates* c_2. Neighborhood-domination implies domination, so we have the following.

Lemma 4. $S + W[4] + B[2] \geq_W S + W[3] + B[2]$.

Proof. In state $S + B[2]$, cells 5 and 6 are Black-captured, so cell 4 White-neighbor-dominates cell 3. □

Lemma 5. $S + W[2] + B[4] \geq_W S + W[6] + B[4]$.

Proof. In state $S + B[4]$, cells 7 and 8 are Black-captured, so cell 2 White-neighbor-dominates cell 6. □

To prove our final lemma, we explicitly construct a second-player strategy for Black on the 4-3-2 carrier.

Lemma 6. $S + W[1] + B[2] \geq_W S + W[6] + B[4]$.

Proof. In state $S + W[6] + B[4]$, Black adopts the following pairing strategy: if White ever occupies one of {2,5}, Black immediately takes the other; Black does this also with {1,3}. Note that cells 7 and 8 are already filled-in for Black due to the edge bridge from $B[4]$. We will show that this pairing strategy always results in a position White-dominated by $S + W[1] + B[2]$.

Note that this pairing strategy maintains the 4-3-2 virtual connection. Thus, via the carrier, White cannot connect cell 1 to either cell 2 or cell 5. Since the pairing strategy prevents White from claiming both cell 2 and cell 5, then the outcome will be that neither is on any minimal White-connector. Thus cells 2 and 5 are captured by Black via this strategy, so White cannot benefit from claiming cell 3, as it is not on any minimal connector. So, without loss of generality we assume White claims cell 1 and Black claims cell 3. But then the outcome of this strategy will be equivalent to $S + W[1] + B[2] + B[3] + B[4] + B[5] + B[6] + B[7] + B[8]$, which is White-dominated by $S + W[1] + B[2]$. Thus, regardless of White's strategy in state $S + W[6] + B[4]$, Black can ensure an outcome that is White-dominated by $S + W[1] + B[2]$. □

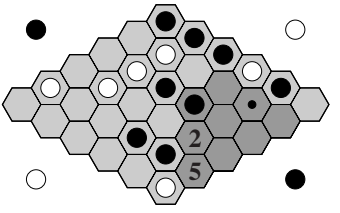


Fig. 9. $S + W[2] + B[4] \not\geq_W S + W[5] + B[4]$

We now prove Theorem 1.

Proof. As mentioned earlier, our assumptions imply that White 4 loses to Black 2. By Lemma 1 and neighbor-domination, $S + W[4] + B[2]$ White-dominates $S + W[3] + B[2]$, $S + W[7] + B[2]$, and $S + W[8] + B[2]$. Thus, White probes $3, 7, 8$ also lose to Black 2.

Likewise, White 1 loses to Black 2 or Black 4. By Lemma 6, $S+W[1]+B[2] \geq_W S + W[6] + B[4]$; by Lemma 1, $S + W[1] + B[4] \geq_W S + W[6] + B[4]$. Thus, regardless of which move defeats White 1, White 6 loses to Black 4. □

Under the hypothesis of Theorem 1, we conjecture that probe 5 is also White-inferior. Our arguments seem unlikely to resolve this, as it is not true for all states S that $S + W[2] + B[4] \geq_W S + W[5] + B[4]$. See Fig. 9.

Any state S in which probe 5 is not White-inferior must satisfy the following conditions: $S' = S + W[6] + B[2] \equiv S + W[5] + B[2]$ (by Corollary 1), so S' wins for White; also, $S + W[2] + B[4]$ loses for White, while $S + W[2] + B[3]$ wins for White (by Lemma 2).

5 Unconditional Pruning of the 4-3-2

Other than not knowing the status of probing at 5, we have so far confirmed our conjecture under the added assumption that Black maintains the 4-3-2. In this section we establish two theorems that apply without making this added assumption.

Theorem 2 applies to a 4-3-2 that lies in an acute corner of the Hex board. Theorem 3 applies to a state which, if it loses for White, implies that seven of the eight 4-3-2 probes also lose.

A Black 4-3-2 can be aligned into an acute corner of the Hex board in two ways, as shown in Fig. 10. When probing such 4-3-2s, the bordering White edge makes capturing easier, yielding the following results.

Lemma 7. *For a Black 4-3-2 as shown in Fig. 10(left), the set of probes* $\{2, 3, 5\}$ *is White-inferior.*

Proof. (sketch) Probes 2 and 5 are capture-dominated by probe 6. Probe 3 can be pruned as follows. First show that $S+W[4] \geq_W S+W[4]+B[1] \equiv S+W[4]+$

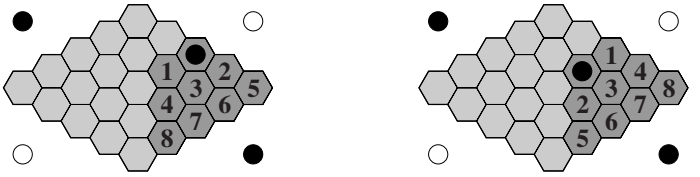


Fig. 10. Acute corner 4-3-2s

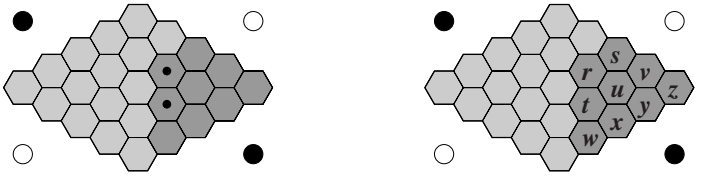


Fig. 11. Dotted cells Black-dominate undotted shaded cells in the acute corner (left). Labels used in the proof of Theorem 2 (right).

$W[3] + B[1]$. It can then be shown that the White probe at 3 is reversible[3] to a Black response at 1, namely that $S \geq_W S + W[3] + B[1]$. From this the desired conclusion follows. We omit the details. □

Lemma 8. *For a Black 4-3-2 as shown in Fig. 10(right), the set of probes {1, 3, 4, 5, 6, 7, 8} is White-inferior.*

Proof. Given any White probe in {1,3,4,5,6,7,8}, Black can respond at cell 2 and capture all cells in the 4-3-2 carrier by maintaining the 4-3-2. □

Theorem 2. *Let S be a Hex state with the nine cells of a potential acute corner Black 4-3-2 all empty, as in Fig. 11. Then each of the seven undotted cells is Black-dominated by at least one dotted cell.*

Proof. (sketch) Let S be a board state in which the nine cells of an acute corner Black 4-3-2, labeled $r, \ldots, z$ as in Fig. 11, are all empty. We want to show that each cell in the carrier is Black-dominated by at least one of r, t.

Cell t Black-capture-dominates cells u, v, w, x, y, z. The argument that cell r Black-dominates cell s is more complex, as follows. Let $S_r = S + B[r]$ and $S_s = S + B[s]$; we want to show that S_r Black-dominates S_s.

First assume that from S_s or S_r, Black is next to move into the carrier. In S_r, a Black move to t Black-captures all other carrier cells, so $S_r + B[t] \geq_B S_s + B[\beta]$ for every possible β in the carrier, so we are done in this case.

Next assume that from S_r or S_s, White is next to move into the carrier. By Lemma 8, from S_r White can do no better than $S_r + W[t]$, so we are done if White has some move from S_s that is at least as good, namely if, for some q in

[3] A P-move is *reversible* if $\overline{P}$ has a response that leaves $\overline{P}$ in at least as good a position as before the P-move. See *Winning Ways, Volume I* [1].

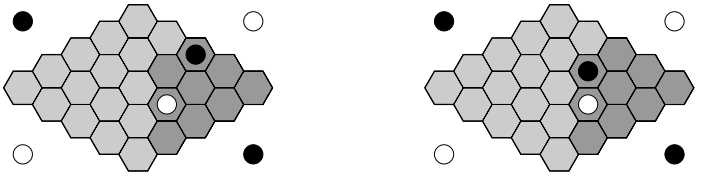


Fig. 12. State $S_s + W[t]$ (left) White-dominates $S_r + W[t]$ (right)

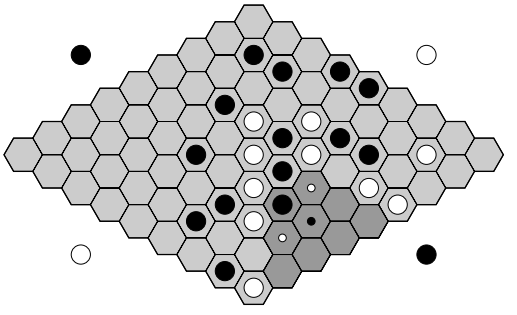


Fig. 13. A state S White-dominated by $S + W[1] + W[2] + B[3]$

the carrier, $S_s + W[q] \geq_W S_r + W[t]$. We can show this for $q = t$; we omit the details. See Fig. 12. □

By Theorem 2, if Black is searching for a winning move and the shaded cells of Fig. 11 are all empty, then Black can ignore the undotted shaded cells.

Next, we consider a result that can be useful when White suspects that probing a particular Black 4-3-2 is futile. We show a state which, if it loses for White, guarantees that seven of the eight probes also lose.

Theorem 3. *Let S be a state with a Black 4-3-2. If $W(S+W[1]+W[2]+B[3])$ is a White loss, then in $W(S)$ each White probe other than 4 loses.*

Proof. $S + W[1] + W[2] + B[3]$ White-dominates both $S + W[1] + B[3]$ and $S+W[2]+B[3]$, so White probes 1, 2 can be pruned. By Lemma 2, $S+W[2]+B[3]$ is equivalent to $S + W[5] + B[3]$, so White 5 can be pruned. Against White probes 3 or 7, strategy decomposition shows that Black wins by replying at cell 4. By Lemma 1 and Corollary 1 respectively, $S + W[3] + B[4]$ White-dominates $S + W[6] + B[4]$, and $S + W[7] + B[4]$ is equivalent to $S + W[8] + B[4]$. □

In terms of being able to prune probes of a 4-3-2, Theorem 3 is useful only if $S' = S + W[1] + W[2] + B[3]$ loses. Not surprisingly, we gain less information about the probes when S' wins. For example, Fig. 13 shows a state S in which White has a winning move from S' but no winning move from S.

6 Conclusions

We have introduced path-domination and neighborhood-domination, two refinements of domination in Hex, and used these notions to find conditions under which probes of an opponent 4-3-2 edge template are inferior moves that can be ignored in the search for a winning move.

In particular, three of the eight probes can be unique winning moves and so cannot in general be discounted; we conjecture that the other five probes are all inferior. Since 4-3-2s arise frequently in Hex, confirming this conjecture would allow significant pruning in solving game states.

We have confirmed the conjecture in various situations. For example, if the player knows that the opponent's immediate reply to a probe will be to restore the template connection, then four of these five remaining probes are inferior.

External conditions might suggest that all probes of a particular 4-3-2 are losing. We have found a state of which the loss implies that seven of the eight probes are losing; establishing this result would allow the seven probes to be ignored.

Also, we have established some domination results that apply when the 4-3-2 lies in an acute corner.

It would be of interest to extend our results to consider the combined maintenance of more than one critical connection, or to automate the inference process so that similar results could be applied to a more general family of virtual connections.

Acknowledgments. The authors thank the referees and the University of Alberta's Hex and GAMES group members, especially Broderick Arneson, for their feedback and constructive criticisms. The authors gratefully acknowledge the support of AIF, iCORE, and NSERC.

References

1. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for Your Mathematical Plays, vol. 1. Academic Press, London (1982)
2. Björnsson, Y., Hayward, R., Johanson, M., van Rijswijck, J.: Dead Cell Analysis in Hex and the Shannon Game. In: Graph Theory in Paris, pp. 45–60. Birkhäuser (2007)
3. Browne, C.: Hex Strategy – Making the Right Connections. A.K. Peters, Natick, Mass (2000)
4. Gardner, M.: Mathematical Games. Scientific American 197, 145–150 (1957)
5. Hayward, R.: A Note on Domination in Hex (manuscript, 2003), www.cs.ualberta.ca/~hayward/publications.html
6. Hayward, R., van Rijswijck, J.: Hex and Combinatorics. Discrete Mathematics 306(19-20), 2515–2528 (2006)
7. Hayward, R., 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, pp. 261–278. Kluwer Academic Publishers, Dordrecht (2003)
8. Hein, P.: Vil de laere Polygon? Politiken newspaper (December 26, 1942)

9. King, D.: Hex templates (2001-2007), http://www.drking.plus.com/hexagons/hex/templates.html
10. Maarup, I.: Thomas Maarup's Hex page. maarup.net/thomas/hex/ (2005)
11. Nasar, S.: A Beautiful Mind. Touchstone Press (1998)
12. Nash, J.: Some games and machines for playing them. Rand Corp. Tech. Rpt. D-1164 (1952)
13. van Rijswijck, J.: Hex webpage (2006), http://www.javhar.net/hex

The Game of Synchronized Domineering

Alessandro Cincotti and Hiroyuki Iida

School of Information Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{cincotti,iida}@jaist.ac.jp

Abstract. In synchronized games players make their moves simultaneously rather than alternately. Synchronized Domineering is the synchronized version of Domineering, a classic two-player combinatorial game. We present the solutions for all the $m \times n$ boards with $m \leq 6$ and $n \leq 6$. Also, we give results for the $n \times 3$ boards, $n \times 5$ boards, and some partial results for the $n \times 2$ boards. Future research is indicated.

1 Introduction

The game of Domineering, also known as Crosscram and Dominoes, is a typical two-player game with perfect information, proposed around 1973 by Göran Andersson [2,7,8]. The two players, usually denoted by Vertical and Horizontal, take turns in placing dominoes (2×1 tile) on a checkerboard. Vertical is only allowed to place its dominoes vertically and Horizontal is only allowed to place its dominoes horizontally on the board. Dominoes are not allowed to overlap and the first player that cannot find a place for one of its dominoes loses. After a time the remaining space may separate into several disconnected regions, and each player must choose into which region to place a domino.

Berlekamp [1] solved the general problem for $2 \times n$ board for odd n. The 8×8 board and many other small boards were recently solved by Breuker, Uiterwijk and van den Herik [3] using a computer search with an adequate transposition-table scheme. Subsequently, Lachmann, Moore, and Rapaport solved the problem for boards of width 2, 3, 5, and 7 and other specific cases [9]. Finally, Bullock solved the 10×10 board [4].

2 Synchronized Games

Initially, the concept of synchronism was introduced in the games of Cutcake [5] and Maundy Cake [6] in order to study combinatorial games where players make their moves simultaneously.

As a result, in the synchronized versions of these games there exist no zero-games, i.e., games where the winner depends exclusively on the player that makes

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 241–251, 2008.

Table 1. The possible outcomes in Synchronized Domineering

	Horizontal ls	Horizontal ds	Horizontal ws
Vertical ls	$G = VHD$	$G = HD$	$G = H$
Vertical ds	$G = VD$	$G = D$	-
Vertical ws	$G = V$	-	-

the second move. Moreover, there exists the possibility of a draw, which is impossible in a typical combinatorial game. In this work, we continue to investigate synchronized combinatorial games by focusing our attention on the game of Domineering.

In the game of Synchronized Domineering, a general instance and the legal moves for Vertical and Horizontal are defined exactly in the same way as defined for the game of Domineering. There is only one difference: Vertical and Horizontal make their legal moves simultaneously, therefore, dominoes are allowed to overlap if they have a 1×1 tile in common. We note that 1×1 overlap is only possible within a simultaneous move. At the end, if both players cannot make a move, then the game ends in a draw, else if only one player can still make a move, then he/she is the winner.

In Synchronized Domineering, for each player there exist three possible outcomes:

- The player has a *winning strategy* (ws) independently of the opponent's strategy, or
- The player has a *drawing strategy* (ds), i.e., he/she can always obtain a draw in the worst case, or
- The player has a *losing strategy* (ls), i.e., he/she does not have a strategy for winning or for drawing.

Table 1 shows all the possible cases. It is clear that if one player has a winning strategy, then the other player has neither a winning strategy nor a drawing strategy. Therefore, the cases $ws - ws$, $ws - ds$, and $ds - ws$ never happen. As a consequence, if G is an instance of Synchronized Domineering, then we have 6 possible legal cases.

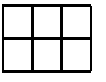
- $G = D$ if both players have a drawing strategy, and the game will always end in a draw under perfect play, or
- $G = V$ if Vertical has a winning strategy, or
- $G = H$ if Horizontal has a winning strategy, or
- $G = VD$ if Vertical can always obtain a draw in the worst case, but he/she could be able to win if Horizontal makes a wrong move, or
- $G = HD$ if Horizontal can always obtain a draw in the worst case, but he/she could be able to win if Vertical makes a wrong move, or
- $G = VHD$ if both players have a losing strategy and the outcome is totally unpredictable.

3 Examples of Synchronized Domineering

The game

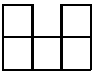
always ends in a draw, therefore $G = D$.

In the game

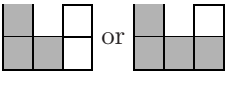


Vertical has a winning strategy moving in the central column, therefore $G = V$.

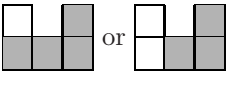
In the game



if Vertical moves in the first column we have two possibilities

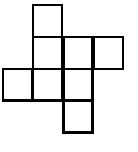


therefore, either Vertical wins or the game ends in a draw. Symmetrically, if Vertical moves in the third column we have two possibilities

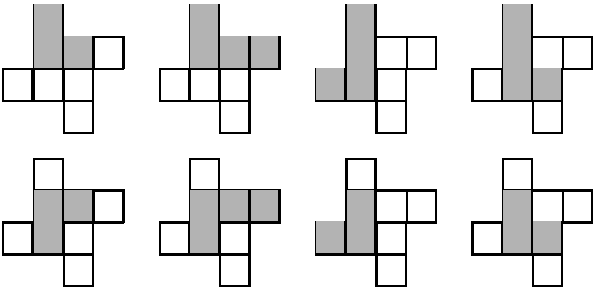


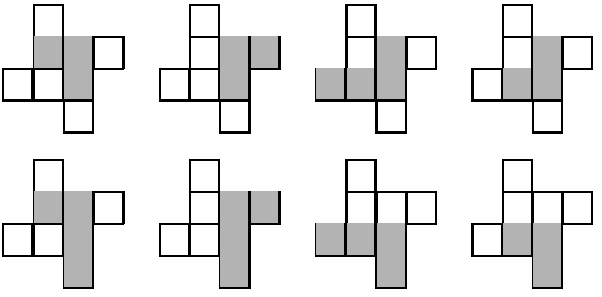
therefore, either Vertical wins or the game ends in a draw. It follows $G = VD$.

In the game



each player has 4 possible moves. The 16 possible outcomes are shown below.





For every move by Vertical (i.e., every row) Horizontal can win or draw (and sometimes lose); likewise, for every move by Horizontal (i.e., every column) Vertical can win or draw (and sometimes lose). As a result it follows that $G = VHD$.

4 Main Results

Table 2 shows the results obtained using an exhaustive search algorithm for the 6×6 board and many other small boards.

Table 2. Outcomes for rectangles in Synchronized Domineering

	1	2	3	4	5	6
1	D	H	H	H	H	H
2	V	D	V	D	V	D
3	V	H	D	H	H	H
4	V	D	V	D	V	D
5	V	H	V	H	D	H
6	V	D	V	D	V	D

Theorem 1. *Let $G = [n, 3]$ be a rectangle of Synchronized Domineering with $n \geq 4$. Then Vertical has a winning strategy.*

Proof. In the beginning, Vertical will always move into the central column of the board, i.e., $(k, b), (k + 1, b)$ where k is an odd number, as shown in Fig. 1. When Vertical cannot move anymore in the central column, let us imagine that we divide the main rectangle into 2×3 sub-rectangles starting from the top of the board (by using horizontal cuts). Of course, if n is odd, then the last sub-rectangle will be of size 1×3, and Horizontal will be able to make one more move. We can classify all these sub-rectangles into 5 different classes.

– Class A. Vertical is able to make two more moves in each sub-rectangle of this class.

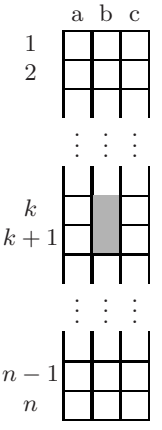
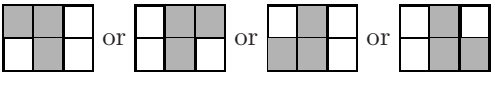
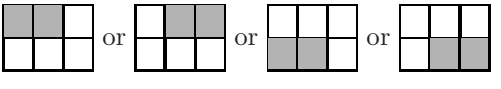


Fig. 1. $G = [n, 3]$

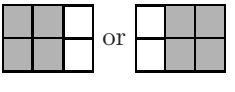
– Class B. Vertical is able to make one more move in each sub-rectangle of this class.



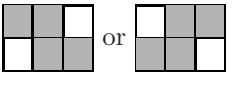
– Class C. Horizontal is able to make one more move in each sub-rectangle and Vertical is able to make at least $\lceil |C|/2 \rceil$ moves where $|C|$ is the number of sub-rectangles belonging to this class. The last statement is true under the assumption that Vertical moves into the sub-rectangles of this class as long as they exist before to move into the sub-rectangles of the other classes.



– Class D. In each sub-rectangle of this class, Horizontal has already made two moves and Vertical is able to make one move.



– Class E. Neither Vertical nor Horizontal are able to make a move in the sub-rectangles of this class.



We show that when Vertical cannot move anymore in the central column, he/she can make a greater number of moves than Horizontal, i.e., $moves(H) < moves(V)$. We denote with $|A|$ the number of sub-rectangles in the A class, with $|B|$ the number of sub-rectangles in the B class, and so on.

Both Vertical and Horizontal have placed the same number of dominoes, therefore

$$|A| = |C| + 2|D| + 2|E|$$

It follows that

$$\begin{aligned} moves(H) &\leq |C| + 1 \\ &= |A| - 2|D| - 2|E| + 1 \\ &\leq |A| + 1 \\ &< 2|A| + |B| + \lceil |C|/2 \rceil + |D| \\ &\leq moves(V) \end{aligned}$$

The condition $|A|+1 < 2|A|+|B|+\lceil |C|/2 \rceil + |D|$ is always true, as shown below.

- If $|A| = 0$ then $|C| = 0$, $|D| = 0$, $|E| = 0$, and $|B| \geq 2$ because by hypothesis $n \geq 4$,
- If $|A| = 1$ then $|C| = 1$, $|D| = 0$, $|E| = 0$,
- If $|A| \geq 2$ then $1 + |A| < 2|A|$.

Theorem 2. *Let $G = [n, 5]$ be a rectangle of Synchronized Domineering with $n \geq 12$. Then Vertical has a winning strategy.*

Proof. In the beginning, Vertical will always move into the columns b and d of the board, i.e., $(k, b), (k + 1, b)$ and $(k, d), (k + 1, d)$, where k is an odd number, as shown in Fig. 2. When Vertical cannot make moves anymore in the columns b and d, let us imagine that we divide the main rectangle into 2×5 sub-rectangles starting from the top of the board (by using horizontal cuts). Of course, if n is odd, then the last sub-rectangle will be of size 1×5 and Horizontal will be

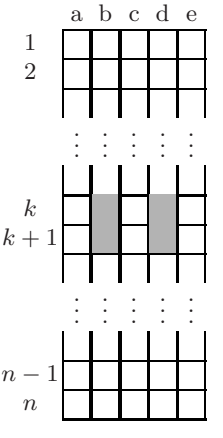


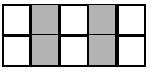
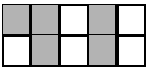
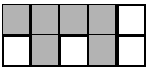
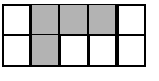
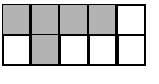
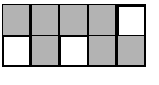
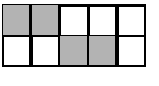
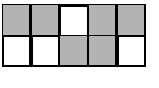
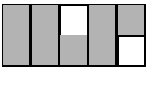
Fig. 2. $G = [n, 5]$

able to make two more moves. We can classify all these sub-rectangles into 10 different classes according to:

- The number of vertical dominoes already placed in the sub-rectangle (vd),
- The number of horizontal dominoes already placed in the sub-rectangle (hd),
- The number of moves that Vertical is able to make in the worst case, in all the sub-rectangles of that class (vm),
- The number of moves that Horizontal is able to make in the best case, in all the sub-rectangles of that class (hm),

as shown in Table 3.

Table 3. The 10 classes for 2×5 sub-rectangles

Class	vd	hd	vm	hm	Example
A	2	0	$3\|A\|$	0	
B	2	1	$2\|B\|$	0	
C	2	2	$\|C\|$	0	
D	1	1	$*$	$\|D\|$	
E	1	2	$*$	$\|E\|$	
F	1	3	0	0	
G	0	2	$*$	$2\|G\|$	
H	0	3	0	$\|H\|$	
I	0	4	0	0	
J	1	2	$\|J\|$	$\|J\|$	

We denote with $|A|$ the number of sub-rectangles in the A class, with $|B|$ the number of sub-rectangles in the B class, and so on. The value of vm in all the sub-rectangles belonging to the classes D, E, and G considered as a group is $|D| + \lceil |D|/2 + |E|/2 + |G|/2 \rceil$. The last statement is true under the assumption that Vertical moves into the sub-rectangles of these classes (D, E, and G) as long as they exist before to move into the sub-rectangles of the other classes.

When Vertical cannot move anymore in the columns b and d, both Vertical and Horizontal have placed the same number of dominoes, therefore

$$2|A| + |B| = |E| + 2|F| + 2|G| + 3|H| + 4|I| + |J| \tag{1}$$

Let us now prove by contradiction that Vertical can make a larger number of moves than Horizontal.

Assume therefore $moves(V) \leq moves(H)$ using the data in Table 3

$$\begin{aligned} 3|A| + 2|B| + |C| + |D| + \\ \lceil |D|/2 + |E|/2 + |G|/2 \rceil + |J| \leq |D| + |E| + 2|G| + |H| + |J| + 2 \end{aligned}$$

and applying Equation 1

$$\begin{aligned} |A| + |B| + |C| + |D| + \\ \lceil |D|/2 + |E|/2 + |G|/2 \lceil + |J| + \\ |E| + 2|F| + 2|G| + 3|H| + 4|I| + |J| \leq |D| + |E| + 2|G| + |H| + |J| + 2 \end{aligned}$$

therefore

$$|A| + |B| + |C| + \lceil |D|/2 + |E|/2 + |G|/2 \rceil + 2|F| + 2|H| + 4|I| + |J| \leq 2$$

which is false because

$$|A| + |B| + |C| + |D| + |E| + |F| + |G| + |H| + |I| + |J| = \lfloor n/2 \rfloor$$

and by hypothesis $n \geq 12$. So $moves(V) \leq moves(H)$ does not hold and consequently $moves(H) < moves(V)$.

By symmetry the following two theorems hold.

Theorem 3. *Let $G = [3, n]$ be a rectangle of Synchronized Domineering with $n \geq 4$. Then Horizontal has a winning strategy.*

Theorem 4. *Let $G = [5, n]$ be a rectangle of Synchronized Domineering with $n \geq 12$. Then Horizontal has a winning strategy.*

Theorem 5. *Let $G = [n, 2]$ be a rectangle of Synchronized Domineering. If n is even then Vertical has a drawing strategy.*

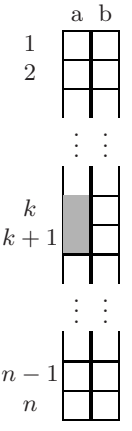
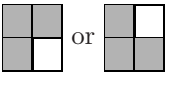


Fig. 3. $G = [n, 2]$

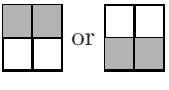
Proof. In the beginning, Vertical will always move into the column a of the board, i.e., $(k, a), (k+1, a)$, where k is an odd number, as shown in Fig. 3. When Vertical cannot move anymore in the column a, let us imagine that we divide the main rectangle into 2×2 sub-rectangles starting from the top of the board (by using horizontal cuts). We can classify all these sub-rectangles into 4 different classes.

- Class A. Vertical is able to make one more move in each sub-rectangle of this class.

- Class B. Neither Vertical nor Horizontal are able to make another move in the sub-rectangles of this class.



- Class C. Horizontal is able to make one more move in each sub-rectangle of this class.



- Class D. In each sub-rectangle of this class Horizontal has already made two moves.

We show that when Vertical cannot move anymore in the column a, he/she can make a greater or equal number of moves compared to Horizontal, i.e., $moves(H) \leq moves(V)$. We denote with $|A|$ the number of sub-rectangles in the class A, with $|B|$ the number of sub-rectangles in the class B, and so on. We observe that $|A| = |C| + 2|D|$ because both Vertical and Horizontal have placed the same number of dominoes. We have

$$\begin{aligned} moves(H) &= |C| \\ &= |A| - 2|D| \\ &\leq |A| \\ &\leq moves(V) \end{aligned}$$

By symmetry the following theorem holds.

Theorem 6. *Let $G = [2, n]$ be a rectangle of Synchronized Domineering. If n is even then Horizontal has a drawing strategy.*

5 Further Research

The previous results suggest the following theorem.

Suggested Theorem 7. *Let G be an $m \times n$ rectangle of Synchronized Domineering. We can distinguish 7 different sub-cases:*

- *If $m = n$, then $G = D$,*
- *If $m > n$ and n is odd, then $G = V$,*
- *If $m > n$, n is even, and m is even, then $G = D$,*
- *If $m > n$, n is even, and m is odd, then $G = H$,*
- *If $m < n$ and m is odd, then $G = H$,*
- *If $m < n$, m is even, and n is even, then $G = D$,*
- *If $m < n$, m is even, and n is odd, then $G = V$.*

This theorem is supported by the previous theorems and the results for small boards, but further efforts are necessary for a formal proof.

Acknowledgment. The authors wish to thank the anonymous referees for helpful suggestions which improved the presentation of the paper. Also, we would like to thank Mary Ann Mooradian for a careful reading of the manuscript.

References

1. Berlekamp, E.R.: Blockbusting and Domineering. Journal of Combinatorial Theory Series A 49(1), 67–116 (1988)
2. Berlekamp, E.R., Conway, J.H., Guy, R.K.: Winning Ways for your Mathematical Plays. Academic Press, San Diego (1982)

3. Breuker, D.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: Solving 8×8 Domineering. Theoretical Computer Science 230(1-2), 195–206 (2000)
4. Bullock, N.: Domineering: Solving Large Combinatorial Search Spaces. ICGA Journal 25(2), 67–84 (2002)
5. Cincotti, A., Iida, H.: The Game of Synchronized Cutcake. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp. 374–379 (2007)
6. Cincotti, A., Iida, H.: The Game of Synchronized Maundy Cake. In: Proceedings of the 7th Annual Hawaii International Conference on Statistics, Mathematics and Related Fields, pp. 422–429 (2008)
7. Conway, J.H.: On Numbers and Games. Academic Press, San Diego (1976)
8. Gardner, M.: Mathematical games. Scientific American 230(2), 106–108 (1974)
9. Lachmann, M., Moore, C., Rapaport, I.: Who Wins Domineering on Rectangular Boards. In: Nowakowski, R.J. (ed.) More Games of No Chance, vol. 42, pp. 307–315. Cambridge University Press, Cambridge (2002)

A Retrograde Approximation Algorithm for Multi-player Can't Stop

James Glenn[1], Haw-ren Fang[2], and Clyde P. Kruskal[3]

[1] Department of Computer Science, Loyola College in Maryland,
4501 N Charles St. Baltimore, MD 21210, USA
jglenn@cs.loyola.edu
[2] Department of Computer Science and Engineering, University of Minnesota,
200 Union St. S.E., Minneapolis, Minnesota, 55455, USA
hrfang@cs.umn.edu
[3] Department of Computer Science, University of Maryland,
A.V. Williams Building, College Park, Maryland 20742, USA
kruskal@cs.umd.edu

Abstract. An n-player, finite, probabilistic game with perfect information can be presented as a $2n$-partite graph. For Can't Stop, the graph is cyclic and the challenge is to determine the game-theoretical values of the positions in the cycles. We have presented our success on tackling one-player Can't Stop and two-player Can't Stop. In this article we study the computational solution of multi-player Can't Stop (more than two players), and present a retrograde approximation algorithm to solve it by incorporating the multi-dimensional Newton's method with retrograde analysis. Results of experiments on small versions of three- and four-player Can't Stop are presented.

1 Introduction

Retrograde analysis has been successfully applied to convergent, deterministic, finite, and two-player zero-sum games with perfect information [10], such as checkers [13] and Awari [12]. In contrast, its application to probabilistic games was generally limited to those with game graph representation being acyclic, such as Yahtzee [6,14] and Solitaire Yahtzee [7]; Pig[1] is a notable exception [11]. We consider the probabilistic games in graph representation with cycles, and are particularly interested in Can't Stop[2]. Our success of tackling one-player and two-player Can't Stop was presented in [8] and [9], respectively. This article presents our study of multi-player Can't Stop that allows more than two

[1] See, e.g., http://en.wikipedia.org/wiki/Pig_(dice).

[2] Can't Stop was designed by Sid Sackson and marketed first by Parker Brothers and now by Face 2 Face Games. It has won a Major Fun award from Majorfun.com and received a Preferred Choice Award from Creative Child Magazine. The rules can be found at http://en.wikipedia.org/wiki/Can't_Stop.

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 252–263, 2008.

players. Our method can also be applied to the multi-player versions of some other probabilistic games, such as Pig, Pig Mania[3], and Hog[4].

An n-player probabilistic game can be represented as a $2n$-partite graph $G = (U_1, \ldots, U_n, V_1, \ldots, V_n, E)$, where U_i corresponds to random events and V_i corresponds to deterministic events for the ith players for $i = 1, \ldots, n$, and $E = (\bigcup_{i=1}^{n}(U_i \times V_i)) \cup (\bigcup_{i=1}^{n} V_i \times \bigcup_{i=1}^{n} U_i)$. In some games, such as Can't Stop, the graph representation is cyclic, which causes difficulty in designing a bottom-up retrograde algorithm. In this article we give a retrograde approximation algorithm to solve n-player Can't Stop, by incorporating the n-dimensional Newton's method into a retrograde algorithm. This indeed is a generalization of the method for two-player Can't Stop [9,5].

The rest of this paper is organized as follows. Section 2 abstracts multi-player probabilistic games. Section 3 gives a retrograde approximation algorithm to solve multi-player Can't Stop. Section 4 presents the indexing scheme. Section 5 summarizes the results of the experimental tests. Our findings are summarized in Sect. 6.

2 Abstraction of Probabilistic Games

We use a game graph $G = (U_1, \ldots, U_n, V_1, \ldots, V_n, E)$ to represent an n-player probabilistic game ($n \geq 2$), where roll and move positions of the ith player are in U_i and V_i, respectively, for $i = 1, \ldots, n$, and $E = (\bigcup_{i=1}^{n} U_i \times V_i) \cup (\bigcup_{i=1}^{n} V_i \times \bigcup_{i=1}^{n} U_i)$. Each position u is associated with a vector of scores $f(u) = (f_1(u), \ldots, f_n(u)) \in \mathbb{R}^n$, where $f_i(u)$ represents the expected score that the ith player achieves in optimal play from u for $i = 1, \ldots, n$. This mapping is denoted by a function $f : \bigcup_{i=1}^{n} U_i \cup V_i \to \mathbb{R}^n$, which is also called a *database* of the game.

For each non-terminal roll position $u \in \bigcup_{i=1}^{n} U_i$, each outgoing edge (u, v) has a weight $0 < p((u, v)) \leq 1$ indicating the probability that the game in u will change into move position v. Then

$$f(u) = \sum_{\forall v \text{ with } (u,v) \in E} p((u, v)) f(v). \tag{1}$$

In optimal play, each player maximizes locally his score[5]. Consider the move positions in V_i of the ith player. For all non-terminal move positions $v_i \in V_i$,

$$f(v_i) = f(\text{argmax}\{f_i(u) : (v_i, u) \in E\}). \tag{2}$$

In other words, the ith player chooses the move to maximize his score at each move position $v_i \in V_i$. A database f that satisfies both conditions (1) and (2) is called a *solution* to G.

First, we consider (1). In this paper we let $f_i(u) \in [0, 1]$ be the probability that the ith player at position u will win the game in optimal play, although in

[3] See, e.g., `http://en.wikipedia.org/wiki/Pass_the_Pigs`.
[4] See, e.g., `http://en.wikipedia.org/wiki/Pig_(dice)#Rule_Variations`.
[5] We use 'he/his' when both 'she/her' and 'he/his' are possible, respectively.

general it can be from any scoring method. Note that we have no assumption of the number of winners at the end of a game. It can be no winner or multiple winners. If a game always ends with exactly one winner, then $\sum_{i=1}^{n} f_i(w) = 1$ for all $w \in \bigcup_{i=1}^{n} U_i \cup V_i$. If in addition $n = 2$, this model coincides with the zero-sum two-player model presented in [9,5] by setting $f_2(u) = 1 - f_1(u)$.

Then we consider (2). Ambiguity occurs if there is more than one maximizer of $f_i(u)$ that results in a different $f(u)$. In such a case two possible disambiguation rules are listed as follows.

- Take the average of $f(u)$ of all maximizers of $f_i(u)$, which means to choose randomly a maximizer.
- Choose a maximizer according to additional assumptions (e.g., collusion between players).

If some ith player's goal is not to maximize his own score but to attack some jth player, then (2) is replaced by

$$f(v_i) = f(\text{argmin}\{f_j(u) : (v_i, u) \in E\})$$

for $v_i \in V_i$. Ambiguity, if present, can be handled in a similar way stated above.

We illustrate an example in Fig. 1, where $u_1, \bar{u}_1 \in U_1$, $v_1, \bar{v}_1 \in V_1$, $u_2, \bar{u}_2 \in U_2$, $v_2, \bar{v}_2 \in V_2$, $u_3, \bar{u}_3 \in U_3$, and $v_3, \bar{v}_3 \in V_3$. The three terminal vertices are $\bar{u}_1$, $\bar{u}_2$ and $\bar{u}_3$ with position values $f(\bar{u}_1) = (1, 0, 0)$, $f(\bar{u}_2) = (0, 1, 0)$, and $f(\bar{u}_3) =$

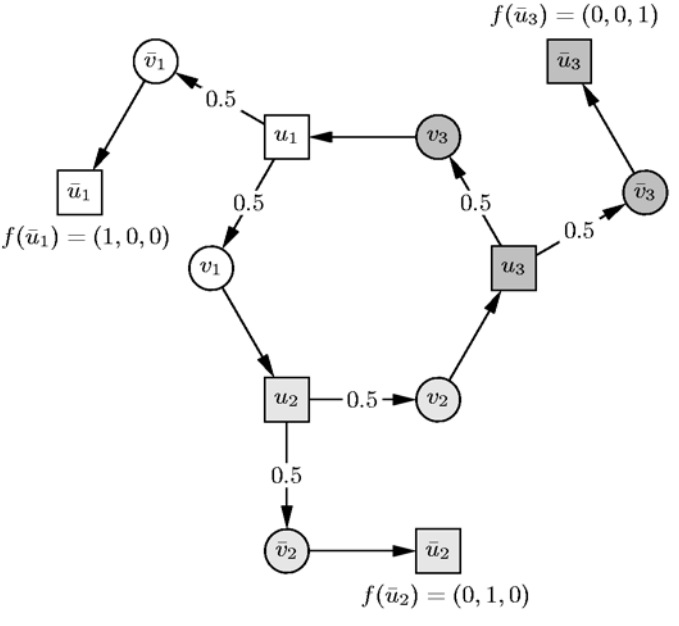


Fig. 1. An example of three-player game graph $G = (U_1, V_1, U_2, V_2, U_3, V_3, E)$

$(0, 0, 1)$. A cycle is formed by the edges between vertices $u_1, v_1, u_2, v_2, u_3, v_3$. This example simulates the last stage of a game of three-player Can't Stop. At position u_1, the first player has 50% chance of winning the game immediately, and a 50% chance of being unable to advance and therefore making no progress at this turn. The second and third players are in the same situation at position u_2 and u_3, respectively. Let $f(u_1) = (x, y, z)$. By (1) and (2),

$$\begin{aligned} f(v_2) = f(u_3) &= (\frac{1}{2}x, \frac{1}{2}y, \frac{1}{2}z + \frac{1}{2}), \\ f(v_1) = f(u_2) &= (\frac{1}{4}x, \frac{1}{4}y + \frac{1}{2}, \frac{1}{4}z + \frac{1}{4}), \\ f(v_3) = f(u_1) &= (\frac{1}{8}x + \frac{1}{2}, \frac{1}{8}y + \frac{1}{4}, \frac{1}{8}z + \frac{1}{8}) = (x, y, z). \end{aligned} \tag{3}$$

Solving the last equation in (3), we obtain $x = \frac{4}{7}$, $y = \frac{2}{7}$ and $z = \frac{1}{7}$, the winning probabilities of the three players when it is the first player's turn to move[6]. This example reveals that solving multi-player Can't Stop is equivalent to solving a system of piecewise linear equations. We give in Sect. 3 an approximation algorithm to solve it by incorporating the multi-dimensional Newton's method with retrograde analysis.

Because of the potential ambiguity of (2), the existence and uniqueness of the solution of multi-player Can't Stop may need further assumptions, and hence are not investigated in this paper.

Note that, however, if the number of players is two (i.e., $n = 2$) and the position value $f(u) = (f(u_1), f(u_2))$ satisfies $f_1(u) + f_2(u) = 1$ for $u \in \bigcup_{i=1}^{2} U_i \cup V_i$ (i.e., always exactly one winner at the end), then no ambiguity of (2) would occur. For two-player Can't Stop we have proved the existence and uniqueness of the solution in [5].

3 Retrograde Solution for Multi-player Can't Stop

Can't Stop is a game for up to four players. It can be generalized to allow even more players. In this section we give a retrograde approximation algorithm for n-player Can't Stop, by incorporating the n-dimensional Newton's method with retrograde analysis. This is generalized from the result of the two-player version game in [9]. We begin with acyclic game graphs for simplicity.

3.1 Game Graph Is Acyclic

For games with acyclic game graphs, such as multi-player Yahtzee, the bottom-up propagation procedure is clear. Algorithm 1 gives the pseudocode to construct the database for an acyclic game graph.

In Algorithm 1, $0^{(n)}$ means a zero vector of size n, and $-\infty^{(n)}$ follows the same syntax. Assuming all terminal vertices are in $\bigcup_{i=1}^{n} U_i$, the set S_2 is initially

[6] Another small example of simplified Parcheesi can be found in [3, Chapter 3].

Algorithm 1. Construct database f for an acyclic game graph

Require: $G = (U_1, V_1, \ldots, U_n, V_n, E)$ is acyclic.
Ensure: Program terminates with (1) and (2) satisfied.
  $\forall u \in \bigcup_{i=1}^{n} U_i$, $f(u) \leftarrow 0^{(n)}$. ▷ Initialization Phase
  $\forall v \in \bigcup_{i=1}^{n} V_i$, $f(v) \leftarrow -\infty^{(n)}$.
  $S_1 \leftarrow$ {terminal positions in $\bigcup_{i=1}^{n} U_i$}
  $S_2 \leftarrow$ {terminal positions in $\bigcup_{i=1}^{n} V_i$} ▷ (†)
  $\forall w \in S_1 \cup S_2$, set $f(w)$ to be its position value.
  repeat ▷ Propagation Phase
    for all $u \in S_1$ and $(v, u) \in E$ **do**
      Determine i such that $v \in V_i$.
      $f(v) \leftarrow f(\text{argmax}\{f_i(w) : (v, w) \in E\})$
      if all children of v are determined **then** ▷ (*)
        $S_2 \leftarrow S_2 \cup \{v\}$
      end if
    end for
    $S_1 \leftarrow \emptyset$
    for all $v \in S_2$ and $(u, v) \in E$ **do**
      $f(u) \leftarrow f(u) + p((u, v))f(v)$
      if all children of u are determined **then** ▷ (**)
        $S_1 \leftarrow S_1 \cup \{u\}$
      end if
    end for
    $S_2 \leftarrow \emptyset$
  until $S_1 \cup S_2 = \emptyset$

empty and (†) is not required. However, it is useful for the reduced graph $\hat{G}$ in Algorithms 2 and 3. We say a vertex is *determined* if its position value is known. By (1) and (2), a non-terminal vertex cannot be determined until all its children are determined. The sets S_1 and S_2 store all determined but not yet propagated vertices. A vertex is removed from them after it is propagated. The optimal playing strategy is clear: given $v \in V_i$, a position with the ith player to move, always make the move (v, u) that maximizes $f_i(u)$.

In an acyclic graph, the *level* (the longest distance to the terminal vertices) for each vertex is well-defined. In Algorithm 1, the position values are uniquely determined level by level. Hence a solution exists. A non-terminal position has its value determined whenever all its children are determined at (*) and (**) in Algorithm 1. One can use a boolean array to trace the determined positions in the implementation. The uniqueness of the solution is subject to the disambiguation rule for (2). If all terminal positions w satisfy $\sum_{i=1}^{n} f_i(w) = 1$, so do all the positions, by recursively applying (1) and (2).

In Algorithm 1, an edge (u, v) can be visited as many times as the out-degree of u because of (*) and (**). The efficiency can be improved as follows. We associate each vertex with a number of undetermined children, and decrease the value by one whenever a child is determined. A vertex is determined after the number is decreased down to zero. As a result, each edge is visited only once

and the algorithm is linear. This is called the *children counting* strategy. For games like Yahtzee, the level of each vertex, the longest distance to the terminal vertices, is known *a priori*. Therefore, we can compute the position values level by level. Each edge is visited only once without counting the children. Note that Algorithm 1, is related to dynamic programming. See, for example, [1] for more information.

3.2 Game Graph Is Cyclic

If we apply Algorithm 1 to a game graph with cycles, then the vertices in the cycles cannot be determined. A naive algorithm to solve the game is described as follows. Given a cyclic game graph $G = (U_1, V_1, \ldots, U_n, V_n, E)$, we prune some edges so the resulting $\hat{G} = (U_1, V_1, \ldots, U_n, V_n, \hat{E})$ is acyclic, and then solve $\hat{G}$ by Algorithm 1. The solution to $\hat{G}$ is treated as the initial estimation for G, denoted by function $\hat{f}$. We approximate the solution to G by recursively updating $\hat{f}$ using (1) and (2). If $\hat{f}$ converges, it converges to a solution to G. The pseudocode is given in Algorithm 2.

Algorithm 2. A naive algorithm to solve a cyclic game graph

Ensure: If $\hat{f}$ converges, it converges to a solution to $G = (U_1, V_1, \ldots, U_n, V_n, E)$.
  Obtain an acyclic graph $G = (U_1, V_1, \ldots, U_n, V_n, \hat{E})$, $\hat{E} \subset E$. ▷ Estimation Phase
  Compute the solution $\hat{f}$ to $\hat{G}$ by Algorithm 1. ▷ (†)
  Use $\hat{f}$ as the initial guess for G.
  $S_1 \leftarrow \{$terminal positions of $\hat{G}$ in $\bigcup_{i=1}^{n} U_i\}$.
  $S_2 \leftarrow \{$terminal positions of $\hat{G}$ in $\bigcup_{i=1}^{n} V_i\}$.
  repeat ▷ Approximation Phase
    for all $u \in S_1$ and $(v, u) \in E$ **do**
      Determine i such that $v \in V_i$.
      $\hat{f}(v) \leftarrow f(\text{argmax}\{\hat{f}_i(w) : (v, w) \in E\})$. ▷ (*)
    end for
    $S_1 \leftarrow \emptyset$
    for all $v \in S_2$ and $(u, v) \in E$ **do**
      $\hat{f}(u) \leftarrow \sum_{\forall w \text{ with } (u,w)\in E} p((u, w))\hat{f}(w)$ ▷ (**)
      $S_1 \leftarrow S_1 \cup \{u\}$
    end for
    $S_2 \leftarrow \emptyset$
  until $\hat{f}$ converges.

An example is illustrated by solving the game graph in Fig. 1. We remove (v_3, u_1) to obtain the acyclic graph $\hat{G}$, and initialize the newly terminal vertex v_3 with position value $(1, 0, 0)$. The solution for $\hat{G}$ has $\hat{f}(u_1) = (\frac{5}{8}, \frac{1}{4}, \frac{1}{8})$. The update is repeated with $\hat{f}(u_1) = (\frac{5}{8}, \frac{1}{4}, \frac{1}{8}), (\frac{37}{64}, \frac{9}{32}, \frac{9}{32}), \ldots, (\frac{4\cdot 8^k+3}{7\cdot 8^k}, \frac{2(8^k-1)}{7\cdot 8^k}, \frac{8^k-1}{7\cdot 8^k}), \ldots,$ which converges to $(\frac{4}{7}, \frac{2}{7}, \frac{1}{7})$, the correct position value of u_1. Therefore $\hat{f}$ converges to the solution to G. Let $e_i(k)$ be the magnitude difference between $\hat{f}_i(u_1)$ at the kth step and its converged value; then $\frac{e_i(k+1)}{e_i(k)} = \frac{1}{8}$ for $i = 1, 2, 3$. Hence it

converges linearly. This naive Algorithm 2 is related to value iteration. See, for example, [2] for more information.

For computational efficiency, we split a given game graph into strongly connected components, and consider the components in bottom-up order. For multi-player Can't Stop, each strongly connected component consists of all the positions with a certain placement of the squares and various placement of the at most three neutral markers for the player on the move. The roll positions with no marker are the *anchors* of the component. When left without a legal move, the game goes back to one of the anchors, and results in a cycle. The outgoing edges of each non-terminal component lead to the anchors in the supporting components. The terminal components are those in which some player has won three columns. Each terminal component has only one vertex with position value in the form $(0, \ldots, 0, 1, 0, \ldots, 0)$; the ith entry is 1 if the i player wins, and otherwise it is 0.

Denote by a cyclic game graph $G = (U_1, V_1, \ldots, U_n, V_n, E)$ a non-terminal component of multi-player Can't Stop and its outgoing edges to the supporting components. Let G_i be the subgraph of G induced by $U_i \cup V_i$ for $i = 1, \ldots, n$. The following two properties hold.

P1. All the graphs G_i for $i = 1, \ldots, n$ are acyclic.

P2. There exist $w_i \in U_i$, such that the edges from G_i to the other vertices (i.e., not in $U_i \cup V_i$) all end at w_{i+1} for $i = 1, \ldots, n$, where we have defined $w_{n+1} \equiv w_1$ for notational convenience.

Properties (P1) and (P2) also hold in some other probabilistic games in strongly connected components, such as Pig, Pig Mania, and Hog. Therefore, the following discussion and our method are applicable to these games.

Let $\hat{G}_i = (U_i \cup \{w_{i+1}\}, V_i, E_i)$ be the induced bipartite subgraph of G for $i = 1, \ldots, n$. By property (P1), $\hat{G}_i$ is acyclic. All the terminal vertices in $\hat{G}_i$ other than w_{i+1} are also terminal in G. By property (P2), the union of $\hat{G}_i$ for $i = 1, \ldots, n$ forms G. Let x_{i+1} be the estimated position value of w_{i+1}. Here $x_{n+1} \equiv x_1$ because of notational convenience $w_{n+1} \equiv w_1$. We can construct a database for $\hat{G}_i$ with x_{i+1} by Algorithm 1. Denote by $\hat{g}_i(x_{i+1}, w)$ the position value of $w \in U_i \cup V_i$ that depends on x_{i+1}. Given $x_2, \ldots, x_{n+1}$, the values of $\hat{g}_i(x_{i+1}, w)$ for $w \in U_i \cup V_i$, $i = 1, \ldots, n$ constitute a solution to G, if and only if

$$\hat{g}_i(x_{i+1}, w_i) = x_i, \qquad i = 1, \ldots, n. \tag{4}$$

The discussion above suggests to solve the system of equations (4) directly. An example is illustrated with the game graph in Fig. 1 as follows. We treat u_1, u_2, u_3 as the three anchors w_1, w_2, w_3, and let x_1, x_2, x_3 be the initial estimate of the position values of them, respectively. The equations (4) are

$$\begin{aligned} x_3 &= 0.5x_1 + (0, 0, 0.5); \\ x_2 &= 0.5x_3 + (0, 0.5, 0); \\ x_1 &= 0.5x_2 + (0.5, 0, 0). \end{aligned} \tag{5}$$

The solution of this linear system is $x_1 = (\frac{4}{7}, \frac{2}{7}, \frac{1}{7})$, $x_2 = (\frac{1}{7}, \frac{4}{7}, \frac{2}{7})$ and $x_3 = (\frac{2}{7}, \frac{1}{7}, \frac{4}{7})$, which are the exact position values of u_1, u_2 and u_3, respectively.

Solving (4) by propagation in value corresponds to the fixed point iteration for computing fixed points of functions[7], and therefore linear convergence can be expected. In contrast, solution (3) inspires us to propagate in terms of the position value x_1 of w_1. The resulting method corresponds to the n-dimensional Newton's method (see, e.g., [4, Chapter 5]). The pseudocode is given in Algorithm 3.

Algorithm 3. An efficient algorithm to solve a cyclic game graph

Require: $G = (U_1, V_1, \ldots, U_n, V_n, E)$ satisfies properties (P1) and (P2).
Ensure: If $\hat{f}$ converges, it converges to a solution to G in the rate of Newton's method.
{Estimation Phase:}
Denote the induced subgraphs $\hat{G}_i = (U_i \cup \{w_{i+1}\}, V_i, E_i)$ for $i = 1, \ldots, n$; $w_{n+1} \equiv w_1$.
{Note that all $\hat{G}_i$ are acyclic and $\bigcup_{i=1}^{n} E_i = E$}.
Estimate the position values of anchors $w_i \in U_i$, denoted by x_i for $i = 1, \ldots, n$.
{Approximation Phase:}
Estimate the position value of w_{n+1} (i.e., w_1); denote it by x_1.
repeat
  for all $i = n, n-1 \ldots, 1$ **do**
    Solve $\hat{G}_i$ based on x_{i+1} in terms of x_1 by Algorithm 1.
    {Propagation by (1) and (2) is done in terms of x_1.}
  end for
  {We have the position value of w_1 in $\hat{G}_1$ in terms of x_1, denoted by $h(x_1)$.}
  Solve $h(x_1) = x_1$ for new estimate x_1.
until it converges (i.e., x_1 is unchanged in value).

Note that Newton's method needs only one iteration to solve a linear system. Indeed, solution (3) is an illustration of applying Algorithm 3 to solve the small example (5), where (x, y, z) in (3) plays the role of x_1 in Algorithm 3. In this case we obtain the solution by one iteration, since the system (5) is linear. In practice, however, the equations (4) are piecewise linear. Hence, multiple iterations are expected to reach the solution. In the experiments on simplified versions of three- and four-player Can't Stop, it always converged, although Newton's method does not guarantee convergence in general.

Consider Algorithm 3. In the estimation phase, the better the initial estimated position value x_1 of the anchors $w_1 (\equiv w_{n+1})$, the fewer iterations are needed to reach the solution. Assuming the game always has exactly one winner at the end, we may reduce one dimension by setting $f_n(u) = 1 - \sum_{i=1}^{n-1} f_i(u)$ for all positions $u \in \bigcup_{i=1}^{n} U_i \cup V_i$. This change of Algorithm 3 results in a method corresponding to the $(n-1)$-dimensional Newton's method.

In Algorithm 3, the graphs $\hat{G}_1, \ldots, \hat{G}_n$ are disjoint except for the anchors $w_1, \ldots, w_n$. Therefore, in the estimation phase, we may initialize the position values $x_1, \ldots, x_n \in \mathbb{R}^n$ of $w_1, \ldots, w_n$. In the approximation phase, we propagate

[7] See, for example, `http://en.wikipedia.org/wiki/Fixed_point_iteration`.

for $\hat{G}_1, \ldots, \hat{G}_n$ in terms of $x_2, \ldots, x_{n+1} \in \mathbb{R}^n$ ($x_{n+1} \equiv x_1$), respectively and separately. The resulting algorithm corresponds to the n^2-dimensional Newton's method and is natively parallel on n processors. We illustrate an example by solving the game graph in Fig. 1. The first Newton's step results in the linear system (5) of 9 variables, which leads to the solution of the game graph in one iteration. (Note that here $x_1, x_2, x_3 \in \mathbb{R}^3$.) In general, the equations (4) are piecewise linear and require multiple Newton's iterations to reach the solution.

A more general model is that an n-player game graph G has m anchors $w_1, \ldots, w_m$ (i.e., removing the outgoing edges of $w_1, \ldots, w_m$ results in an acyclic graph), but does not satisfy properties (P1) and (P2). In this model the incorporation of mn-dimensional Newton's method is still possible, but it may not be natively parallel.

4 Indexing Scheme

We use two different indexing schemes for positions in n-player Can't Stop: one for anchors and another for non-anchors. Because we can discard the position values of non-anchors once we have computed the position value of their anchors, speed is more important than space when computing the indices of non-anchors. Therefore, we use a mixed radix scheme like the one used for one-player Can't Stop [8] and two-player Can't Stop [9] for non-anchor positions.

In this scheme, anchors are described by $(x_2^1, \ldots, x_{12}^1, x_2^2, \ldots, x_{12}^n, t)$ where x_c^p represents the position of player p's square in column c, and t is whose turn it is. Components and positions within components are described in the same way, except that since a component includes n anchors that differ only in whose turn it is, t may be omitted when describing a component, and within a component we record the positions of neutral markers as if they are player t's squares (given the tuple describing a component and a tuple describing a position with that component, we can easily determine where the neutral markers are). Therefore, a position within a component $(x_2^1, \ldots, x_{12}^1, x_2^2, \ldots, x_{12}^n, t)$ is $(y_2^1, \ldots, y_{12}^1, y_2^2, \ldots, y_{12}^n, t)$ where, for all c and p, $y_c^p = x_c^p$, except that in at most three locations we may have $y_c^t > x_c^t$ (player t may have advanced the neutral markers in three columns). The y_c^p and t are used as the digits in the mixed radix system. The place value of the t digit is 1. The place value of the y_2^1 digit is $v_2^1 = 2$, and in general $v_c^p = v_{c-1}^p \cdot (1 + l_{c-1})$ if $c > 2$ and $v_c^p = v_{12}^{p-1} \cdot (1 + l_{12})$ if $c = 2$ and $p > 1$, where l_c denotes the length of column c. The index of a position is then $(t-1) + \sum_{p=1}^{n} \sum_{c=2}^{12} (y_c^p \cdot v_c^p)$.

Because the probability database for the anchors is kept, space is an important consideration when indexing anchors. In the variant used in our experiments, an anchor $(x_2^1, \ldots, x_{12}^n, t)$ is illegal if $x_c^p = x_c^{p'} > 0$ for some $p \neq p'$ (players' squares cannot occupy the same location with a column). Because some positions are not valid anchors, the mixed radix system described above does not define a bijection between anchors and any prefix of $\mathbb{N}$. The indexing scheme therefore maps some indices to nonexistent anchors.

Furthermore, once a column is closed, the locations of the markers in that column are irrelevant; only which player won matters. For example, an anchor u with $x_5^1 = 9$ and $x_5^2 = x_5^3 = 0$ also represents the positions with $x_5^2, x_5^3 \in \{1, \ldots, 8\}$ and all other markers in the same places as u. If the probability database is stored in an array indexed using the mixed radix system as for non-anchors, then the array would be sparse: for the official 3-player game, over 99.9% of the entries would be wasted on illegal and equivalent indices.

In order to avoid wasting space in the array and to avoid the structural overhead needed for more advanced data structures, a different indexing scheme is used that results in fewer indices mapping to illegal, unreachable, or equivalent positions.

We write each position as $((x_2^1, \ldots, x_2^n), \ldots, (x_{12}^1, \ldots, x_{12}^n), t)$. Associate with each n-tuple $(x_c^1, \ldots, x_c^n)$ an index z_c corresponding to its position on a list of the legal n-tuples of locations in column c (i.e., on a list of n-tuples $(y_c^1, \ldots, y_c^n)$ such that $y_c^i \neq y_c^j$ unless $y_c^i = y_c^j = 0$ or $i = j$, and if $y_c^i = l_c$ then $y_c^j = 0$ for $j \neq i$). For the three-player games and a column with $l_c = 2$ this list would be $(0,0,0), (0,0,1), (0,1,0), (1,0,0), (0,0,2), (0,2,0), (2,0,0)$. Therefore, the anchor $((0,0,1), (2,0,0), (1,0,0), (0,0,0), \ldots, (0,0,0), 1)$ (that is, the position in which player 3 has a square in the first space of column 2, player 1 has squares two spaces into column 3 and one space into column 4, and it is player one's turn) would be encoded as $(1, 6, 3, 0, \ldots, 0, 1)$. Those z_c and t are then used as digits in a mixed radix system to obtain the index

$$(t-1) + \sum_{c=2}^{12} z_c \cdot 2 \prod_{d=2}^{c-1} T(d),$$

where the $T(d)$ term in the product is the number of legal, distinct tuples of locations in column d; $T(d) = n + \sum_{z=0}^{n} \binom{n}{z} P(l_d - 1, z)$. The list of n-tuples used to define the z_i's can be constructed so that if component u is a supporting component of v then the indices of u's anchors are greater than the indices of v's and therefore we may iterate through the components in order of decreasing index to avoid counting children while computing the solution.

There is still redundancy in this scheme: when columns are closed, what is important is which columns have been closed and the total number won by each player, but not which columns were won by each player. Before executing Algorithm 3 on a component, we check whether an equivalent component has already been solved. We deal with symmetric positions in the same way.

5 Experiments

The official version of three- and four-player Can't Stop has over 10^{25} and 10^{32} components – too many to solve with currently available technology. As proof of concept, we alternatively have solved simplified versions of multi-player Can't Stop. The simplified games use dice with fewer than six sides, may have shorter columns than the official version, and may award a game to a player for completing fewer than 3 columns. Let (p, n, k, c) Can't Stop denote the p-player game

Table 1. Results of solving simple versions of Multi-player Can't Stop

(p, n, k, c)	Components	Positions	Time	P(win)			
				$P1$	$P2$	$P3$	$P4$
$(3, 2, 1, 1)$	13	207	0.375s	1.000	0.000	0.000	
$(3, 2, 2, 1)$	340	7,410	1.72s	0.804	0.163	0.0332	
$(3, 2, 3, 1)$	6,643	176,064	14.7s	0.717	0.217	0.0657	
$(3, 3, 1, 2)$	74,302	7,580,604	34m45s	0.694	0.230	0.0760	
$(3, 3, 2, 2)$	3,782,833	687,700,305	2d10h	0.592	0.277	0.130	
$(4, 3, 1, 1)$	48,279	2,168,760	19m30s	0.920	0.0737	0.00591	0.000474

played with n-sided dice and columns of length $k, k{+}2, \ldots, k{+}2(n{-}1), \ldots, k$ that is won when a player completes c columns.

We have implemented Algorithm 3 in Java and solved (p, n, k, c) Can't Stop for six combinations of small values of p, n, k, and c. Note that, in all cases, if $n = 2$ then $c = 1$ and if $n = 3$ then $c = 2$ because if we allow larger values of c then the game may end with no player winning the required number of columns. We used an initial estimate of $(\frac{1}{p}, \ldots, \frac{1}{p})$ for the position values of the anchors within a component. We assume that, when a player has a choice of two or more moves that would maximize his expected score but would have different effects on the other players, he makes the same choice each time; exactly which choice is made is determined arbitrarily by the internal ordering of the moves.

Table 1 shows, for the six examined versions of the game, the size of the game graph, the time it took the algorithm to run, and the probability that the each player wins, assuming that each player plays optimally. The listed totals for components and positions within those components do not include the components that were not examined because of equivalence to other components (for (3,3,2,2) Can't Stop there were 4,539,783 such components).

In the five most simplified versions listed in Table 1, the probability of winning the game in a single turn is so high that the optimal strategy never chooses to end a turn early. Colluding players also have the same strategy: in order to prevent one player from winning it is best in these small versions to try to win straightforwardly on the current turn. Because no player ever ends a turn with partial progress in a column, there is never a question of whether or not to avoid an otherwise desirable column to benefit an ally. For (3,3,2,2) Can't Stop there are a few circumstances in which players should end their turns early; this allows a modest gain from collusion: players two and three can reduce player one's chance of winning by about 0.03%.

6 Our Findings

We used a $2n$-partite graph to abstract an n-player probabilistic game. Given a position u, its position value is a vector $f(u) = (f_1(u), \ldots, f_n(u)) \in [0, 1]^n$, with $f_i(u)$ indicating the winning rate of the ith player for $i = 1, \ldots, n$. We investigated the game of multi-player Can't Stop. To obtain the optimal solution,

we generalized an approximation algorithm from [5,8,9] by incorporating the n-dimensional Newton's method with retrograde analysis. The technique was then used to solve simplified versions of three- and four-player Can't Stop. The official versions of three- and four-player Can't Stop have too many components to solve with currently available technology. It may be possible to find patterns in the solutions to the simplified games and use those patterns to approximate optimal solutions to the official game.

References

1. Bertsekas, D.P.: Dynamic Programming and Optimal Control, 3rd edn., vol. I. Athena Scientific (2005)
2. Bertsekas, D.P.: Dynamic Programming and Optimal Control, 3rd edn., vol. II. Athena Scientific (2007)
3. Binmore, K.: Playing for Real: A Text on Game Theory. Oxford University Press, USA (2007)
4. Dennis, J.E., Schnabel, R.B.: Numerical Methods for Unconstrained Optimization and Nonlinear Equations, USA, SIAM, Philadelphia (1996)
5. Fang, H.-r., Glenn, J., Kruskal, C.P.: Retrograde approximation algorithms for jeopardy stochastic games. ICGA Journal 31(2), 77–96 (2008)
6. Glenn, J.: An optimal strategy for Yahtzee. Technical Report CS-TR-0002, Loyola College in Maryland, 4501 N. Charles St, Baltimore MD 21210, USA (May 2006)
7. Glenn, J.: Computer strategies for solitaire yahtzee. In: IEEE Symposium on Computational Intelligence and Games (CIG 2007), pp. 132–139 (2007)
8. Glenn, J., Fang, H.-r., Kruskal, C.P.: A retrograde approximate algorithm for one-player Can't Stop. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M(J.) (eds.) CG 2006. LNCS, vol. 4630, pp. 148–159. Springer, Heidelberg (2007)
9. Glenn, J., Fang, H.-r., Kruskal, C.P.: A retrograde approximate algorithm for two-player Can't Stop. In: Glenn, J. (ed.) CGW 2007 Workshop, Amsterdam, The Netherlands, pp. 145–156 (2007)
10. 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)
11. Neller, T., Presser, C.: Optimal play of the dice game Pig. The UMAP Journal 25(1), 25–47 (2004)
12. Romein, J.W., Bal, H.E.: Solving the game of Awari using parallel retrograde analysis. IEEE Computer 36(10), 26–33 (2003)
13. Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., Sutphen, S.: Building the checkers 10-piece endgame databases. In: van den Herik, H.J., Iida, H., Heinz, E.A. (eds.) Advances in Computer Games 10. Many Games, Many Challenges, USA,, pp. 193–210. Kluwer Academic Publishers, Boston (2004)
14. Woodward, P.: Yahtzee: The solution. Chance 16(1), 18–22 (2003)

AWT: Aspiration with Timer Search Algorithm in Siguo

Hui Lu and ZhengYou Xia

Department of Computer Science,
Nanjing University of Aeronautics and Astronautics, 210016, China
luhui1984@yahoo.com.cn, zhengyou_xia@yahoo.com

Abstract. Game playing is one of the classic problems of artificial intelligence. The Siguo game is an emerging field of research in the area of game-playing programs. It provides a new test bed for artificial intelligence with imperfect information. To improve search efficiency for Siguo with more branches and the uncertain payoff in the game tree, this paper presents a modified Alpha-Beta Aspiration Search algorithm, which is called Alpha-Beta Aspiration with Timer Algorithm (AWT). The AWT can quickly find a suboptimal payoff (acceptable value) from the game tree by adjusting a window with a timer. The timer is controlled by two parameters (M, N) that vary with the chess-board status of Siguo. Experiments show that AWT achieves the goals of the improvability of time efficiency, although it costs a little more memory and does not lead to the best payoff, but to an acceptable payoff.

1 Introduction

Most successful game-playing programs are based on Alpha-Beta, a simple recursive depth-first minimax search algorithm invented in the late 1950's. There is an exponential gap in the size of trees generated by Alpha-Beta [5,9]. This leads to numerous enhancements to the basic algorithm, including Minimal Window Search [2], Aspiration Search [4], Principal Variation Search (PVS) [6], MTD(f) [7,8] and so on. These enhancements have improved the performance of depth-first minimax search considerably.

Aspiration Search is a widely used enhancement. It increases tree-pruning searches with a smaller search window. However, additional search-window reductions run the risk that the program may not be able to find the best payoff. If the search result lies outside the window, it is necessary to re-search with the right window. In this case, the search efficiency will be lowered. So, how to choose the window is an important problem. It is quite difficult in real game playing.

The Siguo game (for an introduction we refer to [10]) is a popular game in China. It provides a new experimental field for artificial intelligence. As a game with imperfect information, the Siguo game has its own properties. First, the size of the game tree built in Siguo is very large. In 1V1 mode, there are 150 moves at a position on average. This is much larger than in Chess, which has about 35

H.J. van den Herik et al. (Eds.): CG 2008, LNCS 5131, pp. 264–274, 2008.

moves. Second, at the beginning of Siguo the prediction of the type of opponent's piece is not exact and the evaluation of a leaf node is not very precise. In this case, we hope to find quickly a suboptimal payoff instead of the best payoff in order to make the program ready for the middle game. For example, if we must make a choice between achieving (1) the best payoff of 20 in 40 seconds and (2) a payoff of 18 in 10 seconds, we may like to choose the second one.

The size of the initial window has to be accurate assuming that Alpha-Beta Aspiration Search is implemented. However, Alpha-Beta Aspiration with a Timer Algorithm (AWT) does not guess the initial size of the window, but adjusts the window dynamically with timer during the search process. In the Alpha-Beta Aspiration Search Algorithm it holds that if the best payoff lies outside the initial window, a re-search is necessary. Since AWT uses the size of the window that is dynamically adjusted, it does not need a re-search. Experiments show that adjusting the window with a timer may improve the search efficiency. The AWT algorithm is well implemented in a Siguo system [10,11,12].

The remainder of this paper is organized as follows. In Sect. 2 we first discuss the motivation of the AWT algorithm and describe the working principle of the AWT algorithm. We also discuss the efficiency and reliability of the AWT algorithm in detail. Some parameters of the AWT algorithm to be implemented in a Siguo system are proposed. In Sect. 3, we describe several experiments and provide an adequate analysis. Our conclusion is given in Sect. 4 together with future work.

2 AWT (Aspiration with Timer)

2.1 Motivation

Traditional methods for perfect-information games are not sufficient to play the games Bridge and Siguo well. Many people design a heuristic search algorithm for each imperfect-information game on the basis of its specific properties, such as a distinct search in Bridge [3].

In the Siguo game the branches of the game tree are very large (computed by Bud *et al.* [1]). Siguo is divided into two kinds of play mode: 1V1 and 2V2. There are 150 moves in the mode of 1V1 and 90 moves in the mode of 2V2 on average at a position. So, the minimax tree generated in the search is quite large. Siguo is an imperfect-information game: though four players place all of the pieces on the game board, each player is assumed to see only (1) his[1] own pieces, (2) the position of the opponent's pieces, and (3) the position of the partner's pieces. However, players cannot see the types of pieces that the opponent and the partner move. Since players cannot see their opponents' types of pieces, the evaluation of leaf nodes is not precise. At the opening and middle phases of Siguo, no more information about the opponent's pieces can be achieved. Therefore, the payoff of leaf nodes is rather uncertain in the two phases. So, it is not worth to spend plenty of time on searching the best payoff since it is not very certain. It is

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

acceptable to achieve the suboptimal payoff by using a little time at the two phases.

Based on the above discussion, we propose a modified Alpha-Beta Aspiration Search Algorithm, which can achieve a suboptimal payoff by using a dynamical adjusting window mechanism that is based on timer.

2.2 AWT: Aspiration with Timer

In an Alpha-Beta Aspiration Search algorithm, the size of the window is set in advance. To avoid re-search aroused by a mistaken evaluation of the window in the Alpha-Beta Aspiration Search algorithm, we do not guess the initial size of the window and use the window of $(-\infty, +\infty)$. To achieve the goal of causing more cutoffs, we increase the lower bound (alpha) of the window by some velocity. Some denotations are described below.

P: P is denoted as the payoff of a max node that is bigger than the lower bound of the window during the search process.

M: M is the first parameter of adjusting the lower bound, where M>1. If we can achieve the payoff(P) of a max node, with P bigger than the lower bound(alpha) and the timer is not timeout during the search process, the lower bound(alpha) is adjusted to P∗M.

N: N is the second parameter of adjusting the lower bound (alpha), where $1 < N < M$. If we cannot achieve the payoff of the max node that is bigger than the lower bound(alpha) when the timer is not timeout, the lower bound(alpha) is adjusted to P∗N. How to set the M and N parameters is shown in Subsection 2.4.

The pseudo code of the AWT algorithm is shown in Fig. 1. We use an example to introduce the search process of AWT. We assume that M=2, N=1.5 in the AWT algorithm and use Fig. 2 as the game tree. The initial alpha and beta are denoted by $-\infty$ and $+\infty$, respectively. We then arrive at the first payoff of a max node that is bigger than alpha, which is leaf node D (payoff=2). According to the AWT algorithm, we can set P=2 and alpha is updated to be $\mathrm{P} * \mathrm{M} = 2 * 2 = 4$. The timer is set and begins to count. Assuming that the timer is not timeout and the achieved payoff of node E is bigger than alpha, alpha is again updated to be $\mathrm{P} * \mathrm{M} = 6 * 2 = 12$. Then C will return 6 to his parent node B. Subsequently B continues to search with the window $(-\infty, 6)$. AWT is similar to the Alpha-Beta algorithm, and node H is cutoff. Then B returns the value 6 to node A. A continues to search with the window $(2 * 6, +\infty)$. The timer is set and begins to count. We assume that there is no better payoff of a max node that is bigger than alpha when the timer is timeout. According to the mechanism of the AWT algorithm, we update alpha and alpha becomes equal to $\mathrm{P} * \mathrm{N} = 6 * 1.5 = 9$. The timer is reset and begins to count. We search the node I and node J with the window $(9, +\infty)$. The value of node K is greater than alpha, so the window of node J is updated to $(2 * 11, +\infty)$. The timer is set and begins to count. We continue to search the second child of J, node L. The value of node L is smaller than alpha. So, J returns the value of 11 to node I. Node I continues to

```
/*  Windows[]={M,N,1}, Represent the parameter of AWT            */
/*  istime is the symbol of timer                                */

AWT(n,alpha,beta)
{
   if(n==Leaf)  g =eval(n);
   else if(n==MaxNode){
      g= -∞ ;
      c=firstchild(n);
      While (g<beta && c!=NULL){
         x=AWT(c,alpha,beta);
         if(x>alpha){
                  g= x;
          windowsindex=0;
                 alpha=g*windows[windowsindex];
                 dwStart=GetTickCount();
           istime=1;
               }
               else if(istime==1){
                  if((GetTickCount()-dwStart)>=searchtime&&windowsindex!=2){
                     windowsindex= windowsindex+1;
                           alpha=g*windows[windowsindex];
                           if(windowindex!=2)  dwStart=GetTickCount();
                  else istime=0;
             }
                }
                c=nextbrother(c);
             }
   }
   else{
      g= +∞ ;
      c=firstchild(n);
      while (g>alpha && c!=NULL){
         x=AWT(c,alpha,beta);
         if(x<g){ g= x; }
         beta=min(g,beta);
         c=nextbrother(c);
      }
   }
   return g;
}
```

Fig. 1. The pseudo code of AWT algorithm

search with window (9, 11). Then we continue to search, node O will be cutoff. Node I returns the value of 11 to node A. A searches with window $(2 * 11, +\infty)$. The timer is set and begins to count. Q returns 10 to node P. The return value

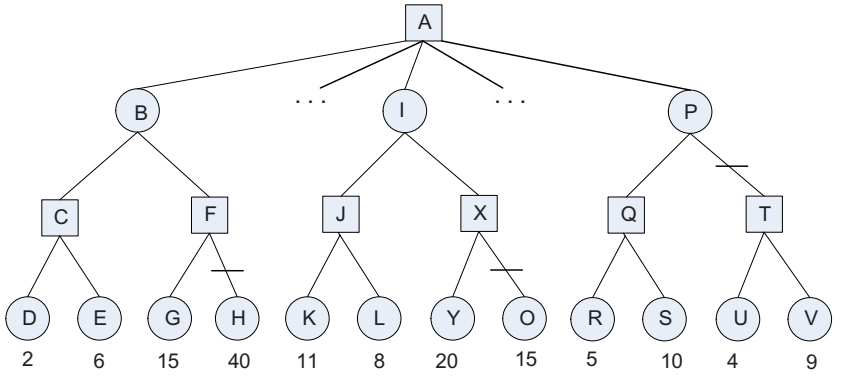


Fig. 2. The example for AWT

is no greater than 22. So it causes a cutoff. The move of node P is no better than node I. So the root A returns the value 11 (node K).

2.3 Efficiency and Reliability of AWT

In this subsection, we discuss the efficiency and reliability of AWT. We use Fig. 3 as an example. In Fig. 3, the Alpha-Beta algorithm visits the nodes in the following order: $A->B->\ ->C->D->E->F->->G->H->I->J$. It finally returns the best payoff of node I (200). All the nodes in Fig. 4 are visited and there is no cutoff in Alpha-Beta.

If we use the AWT algorithm to search the game tree in Fig. 3 (assume M=2, N=1.5), we can achieve the following result: B returns a payoff 40 to its parent A. According to the mechanism of the AWT algorithm, Alpha is updated to $P*M = 40*2 = 80$. If the timer is not timeout when we visit node D, the payoff of node D is 50 and lies out of the window. Node E and F are cut off.

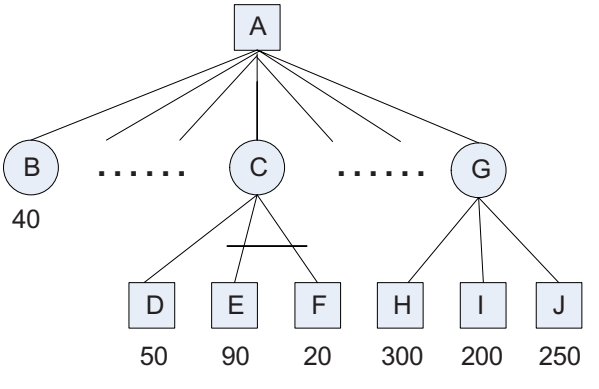


Fig. 3. Example for efficiency of AWT

In contrast, if the timer is timeout when we visit node D, alpha will be updated to be $P * N = 40 * 1.5 = 60$. This window also causes a cutoff because node D lies outside the window. Node E and F are also cut off. It is obvious that AWT seems to cause more cutoffs than the Aspiration algorithm.

In the Alpha-Beta Aspiration Search algorithm, the size of the window (alpha, beta) is set in advance. If the best payoff is just in the window, it will cause more cutoffs than that of the Alpha-Beta algorithm. However, if the best payoff lies outside the window, then all we are told is that a bound on the best payoff is found [5]. To find the true best payoff in this case, a re-search with the right window is necessary. But in AWT, the algorithm starts searching with (-∞, +∞). The window is adjusted dynamically and it has a fast convergent speed. So, AWT will cause more cutoffs than the Aspiration algorithm.

The two parameters M and N in AWT also influence the effect of this algorithm. If the value of M and N is larger, the speed of reducing the window size is faster and this can cause more cutoffs. However, it makes the AWT algorithm miss the best payoff. Of course, AWT can finally achieve the acceptable payoff.

Below we assume that the real best payoff of a game tree is denoted as BESTPAYOFF, and P is denoted as the payoff of max node that is bigger than the lower bound of the window during the search process. FINPAYOFF is the result by which the AWT algorithm completes its search and finally returns the end value. BETA is the upper bound of the window. Five different situations may occur. They will be discussed below.

1. Search with the window (P$*$M, BETA). If BESTPAYOFF is just in this window, the best payoff that is found in AWT is BESTPAYOFF. That is to say: FINPAYOFF = BESTPAYOFF.
2. Search with the window (P$*$M, BETA). If BESTPAYOFF is lower than the lower bound, BESTPAYOFF will never appear in the subsequent search process. That is to say, it misses the best payoff. Then the payoff of AWT is in this bound: BESTPAYOFF/M $<$ FINPAYOFF $\leq$ BESTPAYOFF.
3. Search with the window (P$*$M, BETA). If BESTPAYOFF is lower than the lower bound of the window and it does not find a better payoff when the timer is timeout, the window will extend to (P$*$N, BETA). If BESTPAYOFF appears in the subsequent search process and BESTPAYOFF is in the window (P$*$N, BETA), the best payoff found in AWT will be just that BESTPAYOFF. That is to say: FINPAYOFF = BESTPAYOFF.
4. It does not find a better payoff in the search with window (P$*$N, BETA). BESTPAYOFF lies outside the window and it will never appear in the subsequent search process, in other words it will miss the best payoff. The payoff that is finally found, is this bound: BESTPAYOFF/N $<$ FINPAYOFF $\leq$ BESTPAYOFF.
5. It does not find a better payoff in the search with window (P$*$N, BETA) when the timer is timeout. The window is updated to be (P, BETA). If BESTPAYOFF appears in the subsequent search process, the best payoff will be just that BESTPAYOFF. That is to say: FINPAYOFF = BESTPAYOFF.

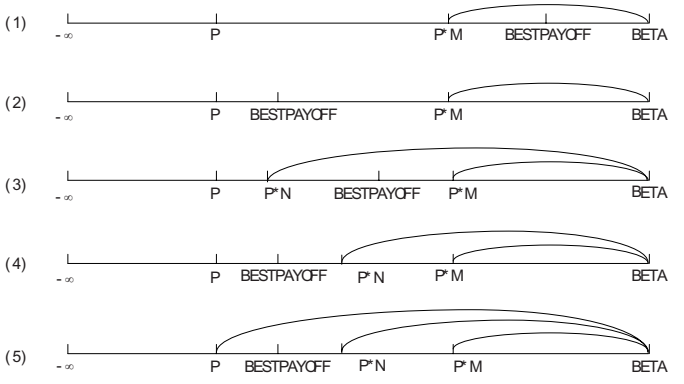


Fig. 4. The reliability of AWT

To clarify the five situations, readers may consult Fig. 4. Using the AWT algorithm, we may miss the best payoff in the previous search process. However, we may achieve this best payoff in the subsequent search process because there are some different leaf nodes that may have the same payoff in the whole game tree. Since the payoff of leaf nodes is evaluated by uncertain information at the opening and middle phases of Siguo, it is acceptable to find a suboptimal payoff when we do not achieve the best payoff. Based on the above discussion and analysis, we may state that the bound of a suboptimal payoff is between BESTPAYOFF/M and BESTPAYOFF. That is: BESTPAYOFF/M $<$ Acceptable value(FINPAYOFF) $\leq$ BESTPAYOFF.

From the above discussion and analysis it is clear that the parameters M and N not only influence the speed of search, but also affect the payoff found in the search. If M and N are larger, more cutoffs are caused. However, the difference between BESTPAYOFF and FINPAYOFF may be very big. Therefore, the M and N parameters of the AWT algorithm are the key factors that influence the algorithm's performance. We discuss their setting below.

2.4 M and N Parameters Setting

At the opening phase, players of Siguo can only obtain rather uncertain information about the opponents. When the game goes on, players may obtain more information. Therefore, the evaluation of the chess board becomes more accurate during playing the game. In our Siguo system, the M and N parameters of the AWT algorithm vary with the information status of the chess board.

We start ordering the twenty-five pieces of an opponent by their positions on the board and label them with nodes $P_i(1 \leq i \leq 25)$ [12]. We let the types of pieces be T = $(t_1, \ldots, t_{12})$, where t_1, t_2, t_3, …, t_{12} are denoted as Sapper, Lieutenant, Captain, …, Bomb, and Flag, respectively. $P_i(T) = (P(t_1)_i, P(t_2)_i, \ldots, P(t_{12})_i)$ is denoted as the type probability distribution of node Pi, where $P(t_1)_i + P(t_2)_i + \ldots + P(t_{12})_i = 1$. Before we discuss how to set M and N, some notations are defined below.

PieceEntroy$_i(s)$: Entropy values of opponent's alive piece on the chess board at turn s, where $1 \leq i \leq 25$ and i is denotes the different pieces.

AverEntroy(s): Average entropy of all pieces of opponent's at turn s. AverEntroy (0) is the initial average entropy of the pieces of the opponent.

Totalnum(s): The number of opponent's alive pieces on the chess board at turn s.

M(s), N(s): the parameters of the AWT algorithm at turn s during playing a game. M(0), N(0) represent the initial parameters.

We use equation 1 to compute the PieceEntroy$_i$(s) of every piece.

$$PieceEntroy_i(s) = -\sum_{j=1}^{12} P_i(t_j) * log(P_i(t_j)) \tag{1}$$

AverEntroy(s) represents the uncertainty of the chess board. If AverEntroy(s) is bigger, the type of opponent's pieces will be more uncertain. In contrast, if AverEntroy(s) is smaller, the prediction of the type of the opponent's pieces will be more exact. We use equation 2 to update AverEntroy(s) of the chess board at turn s.

$$AverEntroy(s) = \frac{\sum_{i=1}^{Totalnum(s)} PieceEntroy_i(s)}{Totalnum(s)} \tag{2}$$

We set the initial value according to our experience, since the type of pieces is uncertain at the beginning of the game. During game play, the player can gradually achieve more information from the opponent, and predict the type of opponent' piece more and more accurately. AverEntroy(s) will become smaller and smaller during the play. In a Siguo system, M and N should be adjusted by the AverEntroy(s) of the board. If the AverEntroy(s) is larger, that is to say the information of opponent's pieces are more uncertain, the value of M(s) and N(s) need to be set larger to improve the speed of search. In contrast, if the information is comparatively rather certain, we should reduce the values of M(s) and N(s) properly to make the payoff approach arrive at the better payoff. According to many experiments performed, we propose to use M(0) = 2, N(0) = 1.1 at the initial setting of parameters. The M(s) and N(s) will be dynamically updated by equation 3 and equation 4, respectively.

$$M(s) = \frac{AverEntroy(s) * M(0)}{AverEntroy(0)}, \quad s \geq 1 \tag{3}$$

$$N(s) = \frac{AverEntroy(s) * N(0)}{AverEntroy(0)}, \quad s \geq 1 \tag{4}$$

3 Experiment and Analysis

In the experiments, we build game trees of Siguo of 7 plies and 30 branches. The evaluation value of the leaf nodes is between -1000 and 1000. The initial search window is (-100,000, +100,000). To assess the different algorithms we use two

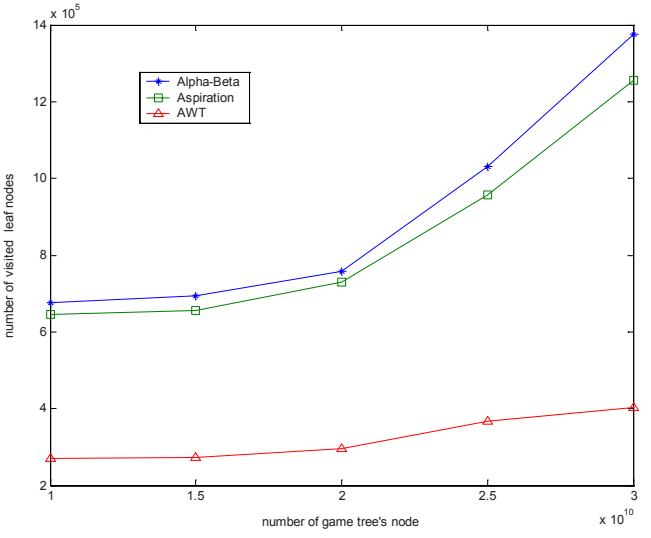


Fig. 5. Comparison of visited nodes in Alpha-Beta, Aspiration and AWT

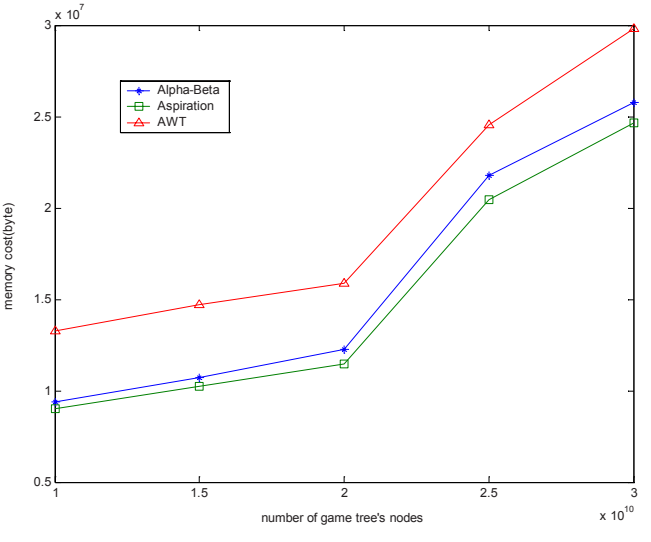


Fig. 6. Comparison of memory in Alpha-Beta, Aspiration, and AWT

parameters: (1) the number of visited leaf nodes and (2) the storage efficiency. We compare Alpha-Beta, Aspiration and AWT by investigating the different sizes of the game trees of Siguo. We found that Alpha-Beta and Aspiration visit more leaf nodes when the size of the game trees increases (see Fig. 5). Aspiration Search explores fewer nodes than the Alpha-Beta Search. However, AWT visits

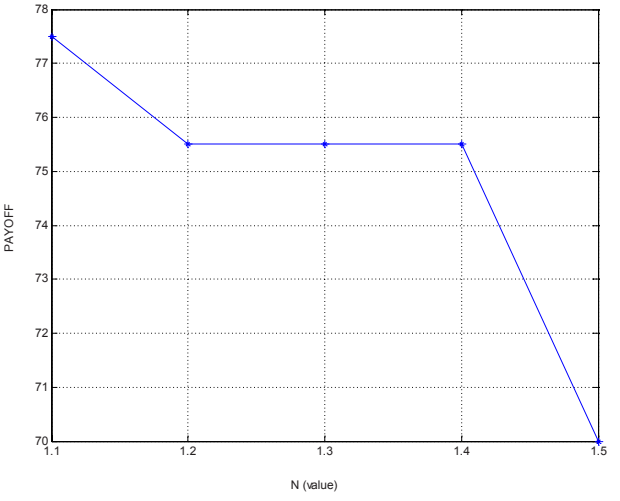


Fig. 7. Performance of AWT for different parameters N

fewer leaf nodes than the Alpha-Beta Search and Aspiration algorithms. The increasing trend is relatively slower.

In Fig. 6, we see that by increasing the size of the game tree, AWT, Alpha-Beta and Aspiration Search need more memory. AWT takes a little more memory than Alpha-Beta and Aspiration Search. The reason that AWT takes more memory is that it spends some memory on dealing with the timer and the window bound.

Actually, we use a game tree of Siguo with $3 * 10^8$ different nodes (1V1 game mode). We investigated the influence of different values of M and N on the payoff that AWT returns after completing the search of this game tree. Before we use the AWT algorithm, we use the Alpha-Beta algorithm to search this game tree and obtain the best payoff of this game tree, which is 84. In Fig. 7, the payoff that uses AWT to return is close to the best payoff (84) with different values of M and N. Under a different value of N, it gets the different payoff. When N is 1.5, payoff is 70. When N is 1.4, 1.3, and 1.2, payoff is 75.7. When N is 1.1, the payoff is 77.5. All the payoff is in the bound [BESTPAYOFF/M, BESYPAYOFF]. The timer is an important parameter. How to set the timer is a big problem. If the value of the timer is too big, the search may omit some good payoff. However, if the value of the timer is set too small, the change of the window has too a high frequency. In our experiments, based on the experience gained previously, we have set the value to 1.5.

4 Conclusion

We start to remark that the opening and middle phases of Siguo, no more information about the opponent's pieces can be obtained. The payoff of the leaf nodes is quite uncertain in these two phases. Therefore, it is not worth to spend much time on searching for the best payoff since the value is not very certain.

In the Alpha-Beta Aspiration Search algorithm, it holds that if the BESTPAYOFF lies outside the initial window (which is set in advance), it is necessary to re-search the whole game tree. In this paper, we have made a small enhancement based on the Aspiration Search algorithm, which is called Aspiration with Timer algorithm (AWT). In AWT, we use a timer to adjust the window dynamically instead of setting an initial window. This can improve the speed of search considerably. However, when we use the AWT algorithm, we may miss the best payoff. However, we will obtain the suboptimal payoff, which is acceptable during play. From the analysis we conclude that our AWT algorithm can guarantee that the suboptimal payoff is between BESTPAYOFF/M and BESTPAYOFF. Experimental results show that the AWT algorithm definitely improves on the speed of Alpha-Beta and Aspiration Search algorithms.

Acknowledgments. This paper is supported by the JiangSu Province Science Technology Foundation under Grant No. BK2006567.

References

1. Bud, A., Albrecht, D., Nicholson, A., Zukerman, I.: Information-Theoretic Advisors in Invisible Chess. In: AI and Statistics 2001, Eighth International Workshop on Artificial Intelligence and Statistics (2001)
2. Campbell, M.S., Marsland, T.A.: A comparison of minimax tree search algorithms. Artificial Intelligence 20(4), 347–367 (1983)
3. Ginsberg, M.L.: GIB: Imperfect information in a computationally challenging game. Journal of Artificial Intelligence Research 14, 303–358 (2001)
4. Kaindl, H., Shams, R., Horacek, H.: Minimax Search Algorithms with and without Aspiration Windows. IEEE Transactions on Pattern Analysis and Machine Intelligence 13(12), 1225–1235 (1991)
5. Knuth, D.E., Moore, R.W.: An analysis of Alpha-Beta Pruning. Artificial Intelligence 6(4), 293–326 (1975)
6. Marsland, T.A.: Relative Efficiency of Alpha-Beta Implementations. In: Proceedings of the 8^{th} International Joint Conference on Artificial Intelligence (IJCAI1983), pp. 763–766 (1983)
7. Plaat, A.: Research Re: search and Re-search. PhD thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, The Netherlands (1996)
8. Plaat, A., Schaeffer, J., Pijls, W., de Bruin, A.: Best-First Fixed-Depth Game-Tree Search in Practice. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 273–281 (1995)
9. Stockman, G.C.: A minimax algorithm better than Alpha-Beta? Artificial Intelligence 12(2), 179–196 (1975)
10. Xia, Z., Zhu, Y., Lu, H.: Using the Loopy Belief Propagation in Siguo. ICGA Journal 30(4), 209–220 (2007)
11. Xia, Z., Hu, Y., Wang, J., Jiang, Y.C., Qin, X.L.: Analyze and Guess Type of Piece in the Computer Game Intelligent System. In: Wang, L., Jin, Y. (eds.) FSKD 2005. LNCS (LNAI), vol. 3614, pp. 1174–1183. Springer, Heidelberg (2005)
12. Xia, Z., Zhu, Y., Lu, H.: Evaluation Function for Siguo Game Based on Two Attitudes. In: The Third International Conference on Fuzzy Systems and Knowledge Discovery, pp. 1322–1331 (2006)

Author Index