Computer Games Workshop 2007

Amsterdam, June 15–17, 2007

MAIN SPONSORS

Preface

We are pleased to present the proceedings of the Computer Games Workshop 2007, Amsterdam, June 15–17, 2007. This workshop will be held in conjunction with the 12th Computer Olympiad and the 15th World Computer-Chess Championship. Although the announcement was quite late, we were pleased to receive no less than 24 contributions. After a "light" refereeing process 22 papers were accepted. We believe that they present a nice overview of state-of-the-art research in the field of computer games.

The 22 accepted papers can be categorized into five groups, according to the type of games used.

Chess and Chess-like Games
In this group we have included two papers on Chess, one on Kriegspiel, and three on Shogi (Japanese Chess).

Matej Guid and Ivan Bratko investigate in *Factors Affecting Diminishing Returns for Searching Deeper* the phenomenon of diminishing returns for additional search effort. Using the chess programs CRAFTY and RYBKA on a large set of grandmaster games, they show that diminishing returns depend on (a) the value of positions, (b) the quality of the evaluation function, and (c) the phase of the game and the amount of material on the board.

Matej Guid, Aritz Pérez, and Ivan Bratko in *How Trustworthy is CRAFTY's Analysis of Chess Champions?* again used CRAFTY in an attempt at an objective assessment of the strength of chess grandmasters of different times. They show that their analysis is trustworthy, and hardly depends on the strength of the chess program used, the search depth applied, or the size of the sets of positions used.

In the paper *Moving in the Dark: Progress through Uncertainty in Kriegspiel* Andrea Bolognesi and Paolo Ciancarini show their latest results on Kriegspiel. This is an incomplete-information chess-variant were the player does not see the opponent's pieces. They tested simple Kriegspiel endings and reveal how their program is able to deal with the inherent uncertainty.

Kosuke Tosaka, Asuka Takeuchi, Shunsuke Soeda, and Hitoshi Matsubara propose in *Extracting Important Features by Analyzing Game Records in Shogi* a method for feature extraction for different groups of Shogi players, using a statistical analysis of game records. They explain how they were able to achieve high discriminant rates in their analysis.

Takeshi Ito introduces the idea of *Selfish Search in Shogi.* This search process mimics human-like play, based on intuition and linear search. He discusses the main characteristics of such a system, and demonstrates its application on some sample Shogi positions.

In *Context Killer Heuristic and its Application to Computer Shogi* Junichi Hashimoto, Tsuyoshi Hashimoto, and Hiroyuki Iida propose a new killer heuristic. Unlike the standard killer heuristic, context killer moves are based on what they call the context-based similarity of positions. Self-play experiments performed in the domain of Shogi demonstrate the effectiveness of the proposed idea.

Go

It is clear that the complex game of Go attracts more and more attention from researchers in Artificial Intelligence. Being one of the hardest traditional games for computers, many research groups undertake the challenge of building strong Go programs and Go tools. Especially intriguing is the success of Monte-Carlo simulations as incorporated in most of the current top programs.

In the paper *Checking Life-and-Death Problems in Go. I: The Program* SCAN-LD, Thomas Wolf and Lei Shen present their program, which checks solutions of life-and-death problems for correctness. After discussing the different types of checks performed by their program, they give statistics resulting from checking a 500-problem Tsume-Go book. They illustrate the mistakes that have been found by examples. A complete list is available on-line.

In *Introducing Playing Style to Computer Go*, Esa A. Seuranen discusses the weaknesses in the current approaches in computer Go and proposes a design, aimed at overcoming some of the shortcomings. In order to achieve this, a position is subdivided into subgames, having local purposes. According to a *playing style* the best move is chosen from the best moves for the subgames.

Tristan Cazenave and Nicolas Jouandeau present in their paper *On the Parallelization of UCT* three parallel algorithms for UCT. They all three improve the results for programs in the field of 9×9 Go.

In *Monte-Carlo Go with Knowledge-Guided Simulations*, Keh-Hsun Chen and Peigang Zhang identify important Go domain knowledge, suited to be used in Monte-Carlo Go. They designed knowledge-guided simulations to be combined with the UCT algorithm, for the 9×9 Go domain. Extensive tests against three top programs demonstrate the merit of this approach.

Rémi Coulom in *Computing Elo Ratings of Move Patterns in the Game of Go* demonstrates another method to incorporate domain knowledge into Go-playing programs. He presents a new Bayesian technique for supervised learning of move patterns from game records, based on a generalization of Elo ratings. Experiments with a Monte-Carlo program show that this algorithm outperforms most previous pattern-learning algorithms.

Jahn-Takeshi Saito, Mark H.M. Winands, Jos W.H.M. Uiterwijk, and H. Jaap van den Herik propose in their paper *Grouping Nodes for Monte-Carlo Tree Search* another idea to enhance Monte-Carlo search in Go programs. They propose to distinguish two types of nodes in a game tree, move nodes and group nodes. A technique, called Alternating-Layer UCT, is designed for managing both types of nodes in a tree consisting of alternating layers of move nodes and group nodes. Self-play experiments show that group nodes can improve the playing strength of a Monte-Carlo program.

Other Abstract Games

Next to the above-mentioned papers dealing with the domains of Chess and Go, six more papers investigate other abstract games.

In *An Efficient Approach to Solve Mastermind Optimally*, the authors Li-Te Huang, Shan-Tai Chen, Shih-Chieh Huang, and Shun-Shii Lin deal with the well-known Mastermind game. They propose a new backtracking algorithm with branch-and-bound pruning (BABBP). This algorithm is more efficient than previous algorithms and can presumably be applied to other games as well.

James Glenn, Haw-ren Fang, and Clyde P. Kruskal in *A Retrograde Approximation Algorithm for Two-Player Can't Stop* investigate the two-player version of Can't Stop, a game designed by Sid Sackson. They present a retrograde approximation algorithm to solve this game. Results of small versions of the game are presented.

Aleksander Sadikov and Ivan Bratko demonstrate *Solving* 20×20 *Puzzles*. They use real-time A* (RTA*) to solve instances of this large version of the well-known sliding-tile puzzles in a reasonable amount of time. Their discovery is based on a recent finding that RTA* works much better with strictly pessimistic heuristics.

In *Reflexive Monte-Carlo Search*, Tristan Cazenave shows that the success of Monte-Carlo methods is not limited to Go, but also can be applied to Morpion Solitaire. Reflexive Monte-Carlo search for the non-touching version breaks the current record and establishes a new record of 78 moves.

Another application of Monte Carlo outside the Go domain is described by François Van Lishout, Guillaume Chaslot, and Jos W.H.M. Uiterwijk in their paper *Monte-Carlo Tree Search in Backgammon*. To their knowledge this is the first application of Monte-Carlo Tree Search to a 2-player game with chance. Preliminary experiments for Backgammon show that their method is suited for on-line learning for evaluating positions, contrary to the top-level Backgammon programs, that are based on off-line learning.

A third application of using Monte Carlo in other games is presented in *The Monte-Carlo Approach in Amazons*, by Julien Kloetzer, Hiroyuki Iida, and Bruno Bouzy. Since Amazons is a game that has the huge branching factor in common with Go, it seemed worthwhile to test Monte Carlo in this domain also. According to experiments the best way is to combine Monte Carlo with a good evaluation function to obtain a high-level program.

Gaming Tools

Besides research on specific abstract computer games presented above, two groups of authors present work on games in a more general framework.

Whereas there has been some work devoted to general gaming engines, these are mainly restricted to complete-information board games. In *Extended General Gaming Model*, Michel Quenault and Tristan Cazenave present an open functional model and its tested implementation. This model extends general gaming to more general games, both card games and board games, both complete- and incomplete-information games, both deterministic and chance games.

Furthermore, Yngvi Björnsson and Jónheidur Ísleifsdóttir introduce in *GTQL: A Query Language for Game Trees* a language specifically designed to query game trees. A software tool based on GTQL helps program developers to gain added insight into the search process, and makes regression testing easier. Experiments show that GTQL is both expressive and efficient in processing large game trees.

Video Games

Whereas abstract games have been in the main stream of AI research already for many years, recently video games started to attract attention in an increasing rate. In the last part of this proceedings we present two papers presenting AI research in the domain of video games.

In *Predicting Success in an Imperfect-Information Game*, Sander Bakkes, Pieter Spronck, Jaap van den Herik, and Philip Kerbusch present their approach for creating an adaption mechanism for automatically transforming domain knowledge into an evaluation function. Experiments are performed in the RTS game SPRING using TD-learning. A term that evaluates tactical positions is added. The results obtained with an evaluation function based on the combination of the defined terms show that the system is able to predict the outcome of a SPRING game reasonably well.

In the final paper, *Inducing and Measuring Emotion through a Multiplayer First-Person Shooter Computer Game*, Paul P.A.B. Merkx, Khiet P. Truong and Mark A. Neerincx developed a database of spontaneous, multi-modal, and emotional expressions. These were based on facial and vocal expressions of emotions uttered by players of a multi-player first-person shooter computer game. The players were then asked to annotate the recordings of their own game-playing. The annotation results revealed interesting insights in current models of emotion, which can be of importance in the development of future video games.

We feel that the papers reproduced in this proceedings give a good representation of current trends in AI research in the domain of computer games. We thank all authors for the generous way to produce and enhance their papers in the very short time available. Finally, we wish all attendants a fruitful Workshop!

The editors

Jaap van den Herik
Jos Uiterwijk
Mark Winands
Maarten Schadd

Organisation

Programme Chairs

Professor H.J. van den Herik
Dr. N.S. Hekster (IBM)
Dr. A. Osseyran (SARA)
Dr. P. Aerts (NCF)

Editors

H.J. van den Herik
J.W.H.M. Uiterwijk
M.H.M. Winands
M.P.D. Schadd

Organising Committee

J.W. Hellemons (chair)
A. Berg
T. van den Bosch
M. den Hartog
H.J. van den Herik
M.P.D. Schadd
J.W.H.M. Uiterwijk
M.H.M. Winands

Sponsors

IBM The Netherlands
SARA Computing and Networking Services
NCF (Foundation of National Computing Facilities)

NWO Physical Sciences (Nederlandse Organisatie voor Wetenschappelijk Onderzoek, Exacte Wetenschappen)
SIKS (School for Information and Knowledge Systems)
BNVKI (Belgian-Dutch Association for Artificial Intelligence)
ChessBase
OnDemand Rentals
MICC-IKAT (Maastricht ICT Competence Centre, Institute for Knowledge and Agent Technology), Universiteit Maastricht

Programme Committee

Table of Contents

Chess and Chess-like Games

Factors Affecting Diminishing Returns for Searching Deeper

Matej Guid[1] and Ivan Bratko[1]

Ljubljana, Slovenia

Abstract. The phenomenon of diminishing returns for additional search effort has been observed by several researchers. We study experimentally additional factors that influence the behaviour of diminishing returns that manifest themselves in *go-deep* experiments. The results obtained on a large set of more than 40,000 positions from chess grandmaster games using programs CRAFTY and RYBKA show that diminishing returns depend on (a) the values of positions, (b) the quality of evaluation function of the program used, and to some extent also on (c) the phase of the game, and the amount of material on the board.

1 Introduction

Deep-search behaviour and diminishing returns for additional search in chess have been burning issues in the game-playing scientific community. Two different approaches took place in the rich history of research on this topic: *self-play* and *go-deep*. While in self-play experiments, two otherwise identical programs are matched with one having a handicap (usually in search depth), go-deep experiments deal with best move changes resulting from different search depths of a set of positions.

The go-deep experiments were introduced for determining the expectation of a new best move being discovered by searching only one ply deeper. The approach is based on Newborn's discovery that the results of self-play experiments are closely correlated with the rate at which the best move changes from one iteration to the next. Newborn [4] formulated the following hypothesis. Let $RI(d + 1)$ denote the rating improvement when increasing search depth from level d to level $d + 1$, and $BC(d)$ the expectation of finding a best move at level d different from the best move found at level $d - 1$, then:

$$RI(d+1) = \frac{BC(d+1)}{BC(d)} \cdot RI(d) \qquad \textbf{(1)}$$

There were some objections about the above equation, e.g. the one by Heinz [1]: »Please imagine a chess program that simply switches back and forth between a few good moves all the time. Such behaviour does surely not increase the playing strength of the program at any search depth.« He suggested that the discovery of »fresh ideas«

[1] Artificial Intelligence Laboratory, Faculty of Computer and Information Science, University of Ljubljana, Slovenia. Email: {matej.guid,bratko}@fri.uni-lj.si.

looks like a much better and meaningful indicator of increases in playing strength than best move change at next iteration of search, and proposes »fresh best« moves instead, defined as new best moves which the program never deemed best before. However, determining $BC(d)$ for higher values of d were consistently used in several experiments. In 1997, Phoenix (Schaeffer [6]) and The Turk (Junghanns et al. [3]) were used to record best-move changes at iteration depths up to 9 plies. In the same year, Hyatt and Newborn [2] let CRAFTY search to an iteration depth of 14 plies. In 1998, Heinz [1] repeated their go-deep experiment with DarkThought. All these experiments were performed on somehow limited datasets of test positions and did *not* provide any conclusive empirical evidence that the best move changes taper off continuously with increasing search depth.

An interesting go-deep experiment was performed by Sadikov and Bratko [5] in 2006. They made very deep searches (unlimited for all practical purposes) possible by concentrating on chess endgames with limited number of pieces. Their results confirmed that diminishing returns in chess exist, and showed that the amount of knowledge a program has influences when diminishing returns will start to manifest themselves.

Remarkable follow-up on previous work done on deep-search behaviour using chess programs was published in 2005 by Steenhuisen [7] who used CRAFTY to repeat the go-deep experiment on positions taken from previous experiments to push the search horizon to 20 plies. He used the same experimental setup to search, among others, a set of 4,500 positions, from the opening phase, to a depth of 18 plies. His results show that the chance of new best moves being discovered decreases exponentially when searching to higher depths, and decreases faster for positions closer to the end of the game. He also reported that the speed with which the best-change rate decreases depends on the test set used.

The latter seems to be an important issue regarding the trustworthiness of the various results obtained by go-deep experiments. How can one rely on statistical evidence from different go-deep experiments, if they obviously depend on the dataset used? In this paper we address this issue, and investigate the hypotheses that the rate at which returns diminish depends on the value of the position. Using a large dataset of more than 40,000 positions taken from real games we conduct go-deep experiments with programs CRAFTY and RYBKA to provide evidence that the chance of new best moves being discovered at higher depths depend on:

1. the values of positions in the dataset,
2. the quality of evaluation function of the program used,

and to some extent also on

3. the phase of the game, and the amount of material on the board.

2 Go-deep Design

Chess programs CRAFTY and RYBKA[2] were used to analyse more than 40.000 positions from real games played in World Championship matches. Each position occurring in these games after move 12 was searched to fixed depth ranging from 2 to 12 plies.

For the measurements done in the go-deep experiments we use the same definitions as provided by Heinz and Steenhuisen. Let $B(d)$ denote the best move after search to depth d, then the following best-move properties were defined:

Best Change	$B(d) \neq B(d - 1)$
Fresh Best	$B(d) \neq B(j)\ \forall j < d$
(d-2) Best	$B(d) = B(d - 2)$ and $B(d) \neq B(d - 1)$
(d-3) Best	$B(d) = B(d - 3)$ and $B(d) \neq B(d - 2)$ and $B(d) \neq B(d - 1)$

We give the estimated probabilities (in %) and their estimated standard errors SE (Equation 2, $N(d)$ stands for the number of observations at search depth d) for each measurement of Best Change. The rates for fresh best, $(d - 2)$ best, and $(d - 3)$ best are given as conditional to the occurrence of a best change. We also provide mean evaluations of positions at each level of search.

$$SE = \sqrt{\frac{BC(d)(1 - BC(d))}{N(d) - 1}} \tag{2}$$

For confidence bounds on the values for best-change rates we use 95%-level of confidence ($\lambda = 1.96$). We use the equation given by Steenhuisen (Equation 3, m represents successes in a sample size of n observations).

$$\frac{m + \frac{\lambda^2}{2} \pm \sqrt{m(1 - \frac{m}{n}) + \frac{\lambda^2}{4}}}{n + \lambda^2} \tag{3}$$

Our hypothesis was the following: best-move changes depend on the value of a given position. It was based on an observation that move changes tend to occur more frequently in balanced positions. To determine the best available approximation of "the true value" of each analysed position, the evaluation at depth 12 served as an oracle. We devised different groups of positions based on their estimated true values.

The rest of the paper is organised as follows: Sections 3 and 4 present the results of go-deep experiments performed by CRAFTY and RYBKA on different groups of positions, based on their estimated true values. In Sections 5 and 6 we observe best-move changes in balanced positions of different groups, based on the phase of the game and the number of pieces on the board. Properties of the groups of positions are described at the beginning of each of these sections. We summarise our results in Section 7.

[2] CRAFTY 19.2 and RYBKA 2.2n2 were used in the experiments.

3 Crafty Goes Deep

Several researchers have used Crafty for their go-deep experiments. However none had such a large set of test positions at disposal. Steenhuisen observed deep-search beahaviour of Crafty on different test sets and reported different best-change rates and best-change rate decreases for different test sets. This and the following section will show that best-change rates strongly depend on the values of positions included in a test set.

We divided the original test set into six subsets, based on the evaluations of positions obtained at depth 12 as presented in Table 1. In usual terms of chess players, the positions of Groups 1 and 6 could be labeled as positions with a "decisive advantage", those of Groups 2 and 5 with a "large advantage", while Groups 3 and 4 consist of positions regarded as approximately equal or with a "small advantage" at most.

Table 1. Subsets with positions of different range of evaluations obtained at level 12 (Crafty).

Group	1	2	3	4	5	6
Evaluation(x)	$x<-2$	$-2\leq x<-1$	$-1\leq x<0$	$0\leq x<1$	$1\leq x<2$	$x\geq 2$
Positions	4,011	3,571	10,169	18,038	6,008	6,203

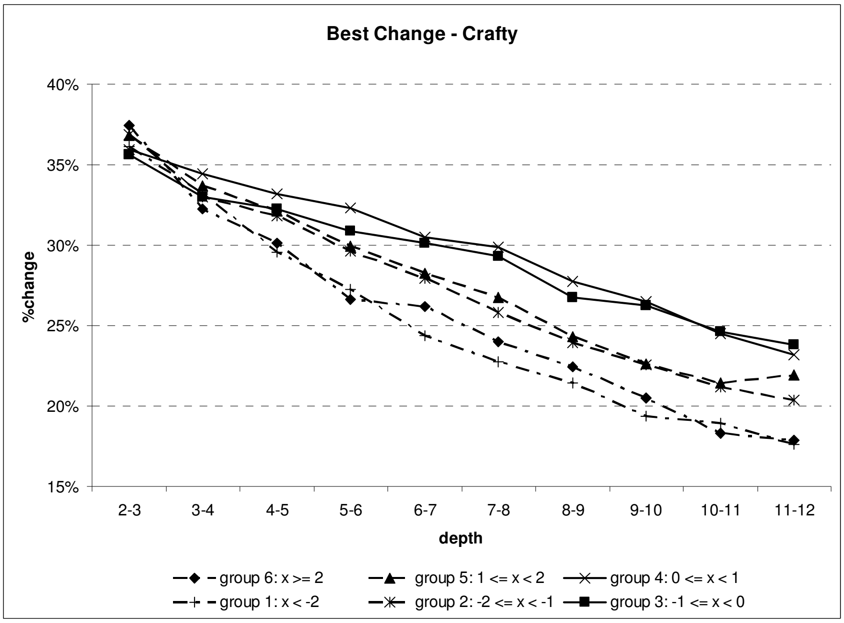


Fig. 1. Go-deep results of Crafty on the six different groups of positions.

The results for each of the six groups are presented in Fig. 1. The curves clearly show different deep-search behaviour of the program for the different groups, depending on the estimated value of positions they consist of. The chance of new best moves being discovered at higher depths is significantly higher for balanced positions than for positions with decisive advantage. It is interesting to observe that this phenomenon does not yet occur at the shallowest search depths, while in the results of RYBKA it manifests itself at each level of search.

The following two tables show the best-move properties for Groups 4 and 6. While the results resemble the ones obtained by Steenhuisen on the 4,500 positions of the ECO test set in a sense that both best change and fresh best rates decrease consistently with increasing search depth, the rates nevertheless significantly differ for each of the two groups of positions.

Table 2. Results of CRAFTY for the 18,038 positions of Group 4.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	35.96 (0.36)	100.00	-	-	0.36
3-4	34.47 (0.35)	74.88	25.12	-	0.37
4-5	33.18 (0.35)	64.16	27.34	8.50	0.37
5-6	32.34 (0.35)	54.38	28.44	11.38	0.37
6-7	30.48 (0.34)	49.53	31.14	9.51	0.37
7-8	29.86 (0.34)	42.81	31.45	11.27	0.38
8-9	27.75 (0.33)	40.02	33.87	10.81	0.38
9-10	26.48 (0.33)	37.77	33.31	10.57	0.38
10-11	24.53 (0.32)	34.79	33.48	11.14	0.38
11-12	23.17 (0.31)	32.26	33.07	12.04	0.39

Table 3. Results of CRAFTY for the 6,203 positions of Group 6.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	37.42 (0.61)	100.00	-	-	2.64
3-4	32.27 (0.59)	73.93	26.07	-	2.76
4-5	30.13 (0.58)	64.85	24.83	10.33	2.84
5-6	26.60 (0.56)	55.70	28.06	9.70	2.95
6-7	26.21 (0.56)	49.88	27.37	10.52	3.04
7-8	23.99 (0.54)	39.92	31.18	11.02	3.17
8-9	22.44 (0.53)	37.21	32.18	12.72	3.29
9-10	20.47 (0.51)	36.30	30.79	11.50	3.42
10-11	18.30 (0.49)	31.37	32.42	12.07	3.54
11-12	17.85 (0.49)	29.27	29.99	13.91	3.68

The 95%-confidence bounds for Best change (calculated using the Equation 2 given in Section 2) at the highest level of search performed for the samples of 18,038 and 6,203 positions of Groups 4 and 6 are [22.56;23.97] and [16.91;18.82], respectively.

4 RYBKA Goes Deep

RYBKA is currently the strongest chess program according to the SSDF rating list [8]. To the best of our knowledge there were no previous go-deep experiments performed with this program. The results in this section will not only confirm that best-change rates depend on the values of positions, but also demonstrate that the chance of new best moves being discovered at higher depths is lower at all depths compared to CRAFTY, which is rated more than 250 rating points lower on the aforementioned rating list.

Table 4. Subsets with positions of different range of evaluations obtained at level 12 (RYBKA).

Group	1	2	3	4	5	6
Evaluation(x)	$x<-2$	$-2\leq x<-1$	$-1\leq x<0$	$0\leq x<1$	$1\leq x<2$	$x\geq 2$
Positions	1,263	1,469	9,808	22,644	3,152	2,133

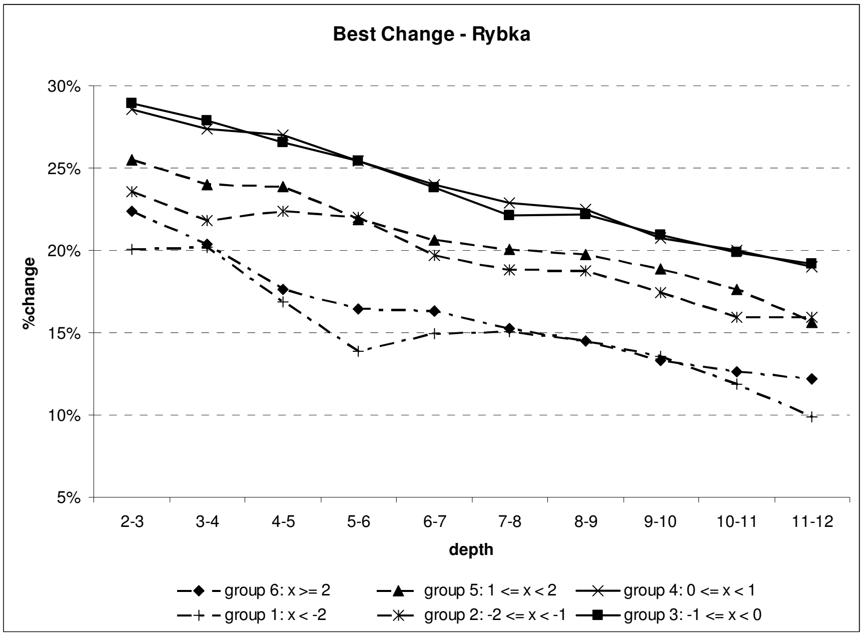


Fig. 2. Go-deep results of RYBKA on the six different groups of positions.

The results of RYBKA presented in Fig. 2 resemble the results of CRAFTY in Fig.1, except that all the curves appear significantly lower on the vertical scale. This result seems to be in line with the observation, based on the results by Sadikov and Bratko, that the amount of knowledge a program has (or the quality of evaluation function) influences deep-search behaviour of a program. The big difference in strength of the two programs is likely to be the consequence of RYBKA having a stronger evaluation function; it is as well commonly known that chess players prefer evaluations of this program to CRAFTY's evaluations. In their study, Sadikov and Bratko claim that diminishing returns will start to manifest themselves earlier using a program with a stronger evaluation function, based on experiments performed on chess endgames, at the same time suspecting that similar results would be obtained with more pieces on the board. The results presented here seem to be in accordance with that conjecture.

Table 5. Results of RYBKA for the 22,644 positions of Group 4.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	28.59 (0.30)	100.00	-	-	0.31
3-4	27.36 (0.30)	71.42	28.58	-	0.31
4-5	27.00 (0.30)	62.95	27.12	9.93	0.31
5-6	25.44 (0.29)	53.32	28.13	10.45	0.31
6-7	24.00 (0.28)	49.91	26.63	11.21	0.30
7-8	22.88 (0.28)	45.78	26.85	11.37	0.30
8-9	22.50 (0.28)	42.97	25.63	11.46	0.30
9-10	20.73 (0.27)	37.17	28.46	11.31	0.30
10-11	20.03 (0.27)	36.16	27.76	11.78	0.30
11-12	19.01 (0.26)	34.08	27.87	11.85	0.30

Table 6. Results of RYBKA for the 2,133 positions of Group 6.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	22.36 (0.90)	100.00	-	-	2.49
3-4	20.39 (0.87)	77.24	22.76	-	2.60
4-5	17.63 (0.83)	66.76	24.20	9.04	2.77
5-6	16.41 (0.80)	54.86	25.43	10.57	2.89
6-7	16.32 (0.80)	49.71	26.44	10.06	3.01
7-8	15.24 (0.78)	44.00	23.69	13.23	3.14
8-9	14.49 (0.76)	45.63	24.60	10.36	3.27
9-10	13.31 (0.74)	42.61	23.94	12.68	3.42
10-11	12.61 (0.72)	37.92	24.16	8.55	3.59
11-12	12.19 (0.71)	36.54	30.00	7.31	3.75

The 95%-confidence bounds for Best change at the highest level of search performed for the samples of 22,644 and 2,133 positions of Groups 4 and 6 are [18.51;19.53] and [10.87;13.65], respectively.

5 Diminishing Returns and Phase of the Game

Steenhuisen was the first to point out that the chance of new best moves being discovered at higher depth decreases faster for positions closer to the end of the game. However, having in mind that deep-search behaviour depends on the values of positions in a test set, it seems worthwhile to check whether his results were just the consequence of dealing with positions with a decisive advantage (at least on average) in a later phase of the game. For the purpose of this experiment we took only a subset with more or less balanced positions with depth 12 evaluation in range between -0.50 and 0.50. Our results show that in the positions that occurred in the games later than move 50, the chance of new best moves being discovered indeed decreases faster, which agrees with Steenhuisen's observations. The experiments in this and the following section were performed by CRAFTY.

Table 7. Six subsets of positions of different phases of the game, with evaluations in range between -0.50 and 0.50, obtained at search depth 12.

Group	1	2	3	4	5
Move no.(x)	x<20	20<x<30	30<x<40	40<x<50	x>50
Positions	7,580	5,316	2,918	1,124	891

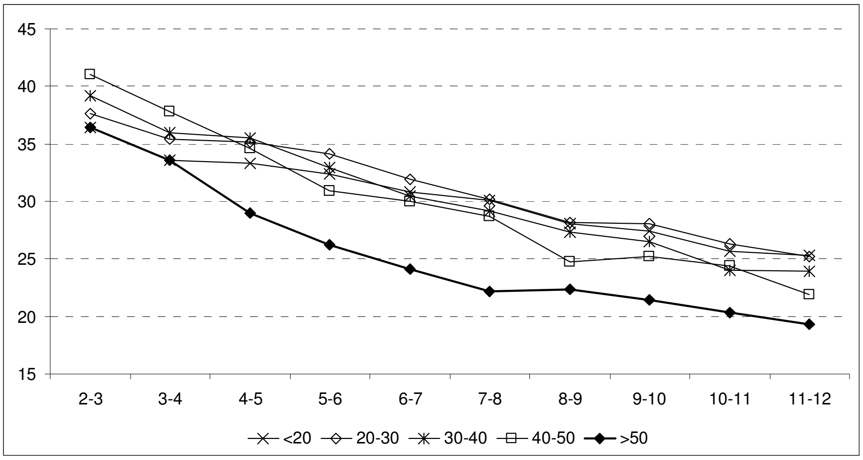


Fig. 3. Go-deep results with positions of different phases of the game.

The results presented in Fig. 3 show that while there is no obvious correlation between move number and the chance of new best moves being discovered at higher depth, in the positions of Group 5 that occurred closer to the end of the game it nevertheless decreases faster than in the positions of other groups.

Table 8. Results for the 7,580 positions of Group 1.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	36.41 (0.55)	100.00	-	-	0.08
3-4	33.63 (0.54)	75.56	24.44	-	0.08
4-5	33.35 (0.54)	63.57	27.73	8.70	0.08
5-6	32.39 (0.54)	54.01	29.74	10.55	0.08
6-7	30.83 (0.53)	49.64	31.79	9.33	0.07
7-8	30.08 (0.53)	44.65	31.49	10.22	0.07
8-9	28.10 (0.52)	41.60	31.64	10.47	0.07
9-10	27.40 (0.51)	37.31	34.28	10.01	0.07
10-11	25.66 (0.50)	35.73	34.91	10.03	0.07
11-12	25.32 (0.50)	31.16	34.55	11.52	0.07

Table 9. Results for the 891 positions of Group 5.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	36.48 (1.61)	100.00	-	-	0.07
3-4	33.56 (1.58)	74.58	25.42	-	0.05
4-5	28.96 (1.52)	60.08	29.07	10.85	0.05
5-6	26.26 (1.48)	50.00	28.21	14.96	0.02
6-7	24.13 (1.43)	46.51	29.77	11.16	0.02
7-8	22.22 (1.39)	46.46	27.78	8.08	0.02
8-9	22.33 (1.40)	35.68	33.17	12.06	0.02
9-10	21.44 (1.38)	38.22	27.75	9.95	0.02
10-11	20.31 (1.35)	33.70	32.60	11.05	0.02
11-12	19.30 (1.32)	26.16	36.63	8.14	0.00

The 95%-confidence bounds for Best change at the highest level of search performed for the samples of 7,580 and 891 positions of Groups 1 and 5 are [24.35;26.31] and [16.85;22.03], respectively.

6 Diminishing Returns and Material

Phase of the game is closely correlated with the amount of material on the board. Therefore, in accordance with previous observations, it could be expected that the rate of best-change properties will be lower in positions with less pieces on the board. The results of this section confirm that with a total value of pieces less than 15 for each of the players, the chance of new best moves being discovered at higher depth decreases faster, albeit only from depth 5 on (also the differences are not so obvious as in the previous section). In the total value of the pieces, the pawns are counted in and for the values of pieces the commonly accepted values are taken (queen = 9, rook = 5, bishop = 3, knight = 3, pawn = 1).

Table 10. Six subsets of positions with different amount of material on the board (each player starts with the amount of 39 points), with evaluations in range between -0.50 and 0.50, obtained at search depth 12.

Group	1	2	3	4	5	6
Material(x)	x<15	15≤x<20	20≤x<25	25≤x<30	30≤x≤35	x>35
Positions	3,236	1,737	2,322	2,612	5,882	4,112

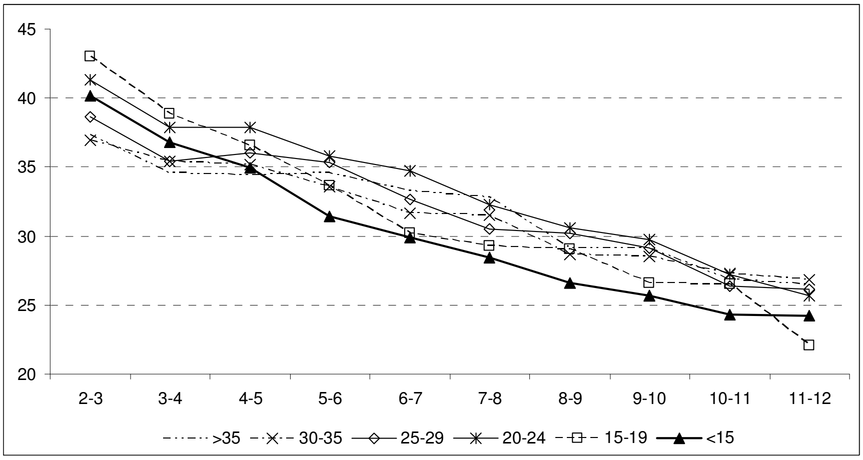


Fig. 4. Go-deep results with positions with different amount of material on the board.

Fig. 4 shows that material and best move changes are not clearly correlated. It is only the curve for positions with the total piece value of less than 15 points of material (for each of the players) that slightly deviate from the others. Surprisingly, we did not spot any significant deviations in positions with even less material (e.g. < 10) either.

Table 11. Results for the 4,112 positions of Group 6.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	37.33 (0.75)	100.00	-	-	0.07
3-4	34.53 (0.74)	74.93	25.07	-	0.07
4-5	34.39 (0.74)	64.85	27.02	8.13	0.07
5-6	34.53 (0.74)	56.06	27.96	10.77	0.07
6-7	33.29 (0.74)	50.55	31.48	8.11	0.07
7-8	32.78 (0.73)	44.96	31.90	9.42	0.07
8-9	29.13 (0.71)	43.32	30.05	10.02	0.07
9-10	29.13 (0.71)	40.65	31.72	9.85	0.07
10-11	26.80 (0.69)	35.84	32.76	11.34	0.07
11-12	26.53 (0.69)	31.71	35.20	11.27	0.07

Table 12. Results for the 3,236 positions of Group 1.

Search depth	Best change in % (SE)	Fresh Best in %	(d-2) Best in %	(d-3) Best in %	mean evaluation
2-3	40.17 (0.86)	100.00	-	-	0.07
3-4	36.80 (0.85)	70.19	29.81	-	0.07
4-5	34.92 (0.84)	60.18	30.88	8.94	0.06
5-6	31.40 (0.82)	49.41	33.17	11.52	0.05
6-7	29.88 (0.80)	46.74	31.33	10.44	0.05
7-8	28.40 (0.79)	42.87	30.36	9.47	0.04
8-9	26.58 (0.78)	35.93	34.53	11.05	0.04
9-10	25.68 (0.77)	34.18	32.13	12.76	0.04
10-11	24.32 (0.75)	32.15	34.18	10.93	0.03
11-12	24.23 (0.75)	30.74	33.80	9.57	0.03

The 95%-confidence bounds for Best change at the highest level of search performed for the samples of 4,112 and 3,236 positions of Groups 6 and 1 are [25.20;27.90] and [22.78;25.73], respectively.

7 Conclusions

Deep-search behaviour and the phenomenon of diminishing returns for additional search effort have been studied by several researchers, whereby different results were obtained on the different datasets used in *go-deep* experiments. In this paper we studied some factors that affect diminishing returns for searching deeper. The results obtained on a large set of more than 40,000 positions from real chess games using programs CRAFTY and RYBKA suggest that diminishing returns depend on:

1. the values of positions in the dataset,
2. the quality of evaluation function of the program used, and also on
3. the phase of the game, and the amount of material on the board.

Among other findings, the results also demonstrated with a high level of statistical confidence that both »best change« and »fresh best« rates (as defined by Newborn [4] and Heinz [1], respectively) decrease with increasing search depth in each of the subsets of the large dataset used in this study.

References

[1] Heinz, E. A. DARKTHOUGHT Goes Deep. *ICCA Journal*, Vol. 21, No. 4, pp. 228-244, 1998.

[2] Hyatt, R. M. and Newborn, M. M. CRAFTY Goes Deep. *ICCA Journal*, Vol. 20, No. 2, pp. 79–86, 1997.

[3] Junghanns, A., Schaeffer, J., Brockington, M. G., Björnsson, Y., and Marsland, T. A. Diminishing Returns for Additional Search in Chess. *Advances in Computer Chess 8* (eds. H. J. van den Herik and J.W. H. M. Uiterwijk), pp. 53–67, Universiteit Maastricht, 1997.
[4] Newborn, M. M. A Hypothesis Concerning the Strength of Chess Programs. *ICCA Journal*, Vol. 8, No. 4, pp. 209-215, 1985.
[5] Sadikov A. and Bratko I. Search versus knowledge revisited again. *Proceedings of the 5th International Conference on Computers and Games (CG2006)*, 2006.
[6] Schaeffer, J. Experiments in Search and Knowledge. Ph.D. thesis, University of Waterloo. Also printed as Technical Report (TR 86-12), Department of Computer Science, University of Alberta, Alberta, Canada, 1986.
[7] Steenhuisen J. R. New results in deep-search behaviour. *ICGA Journal*, Vol. 28, No. 4, pp. 203-213, 2005.
[8] *The SSDF Rating List*: http://web.telia.com/~u85924109/ssdf/list.htm

How Trustworthy is CRAFTY'S Analysis of Chess Champions?

Matej Guid[1], Aritz Pérez[2], and Ivan Bratko[1]

[1]Univ. of Ljubljana, Slovenia, and [2]Univ. of San Sebastián, Spain

Abstract. Guid and Bratko carried out a computer analysis of games played by World Chess Champions as an attempt at an objective assessment of chess playing strength of chess players of different times. Chess program CRAFTY was used in the analysis. Given that CRAFTY's official chess rating is lower than the rating of many of the players analysed, the question arises to what degree that analysis could be trusted. In this paper we investigate this question and other aspects of the trustworthiness of those results. Our study shows that it is not very likely that the ranking of at least the two highest-ranked players would change if (1) a stronger chess program was used, or (2) if the program would search deeper, or (3) larger sets of positions were available for the analysis.

1 Introduction

The emergence of high-quality chess programs provided an opportunity of a more objective comparison between chess players of different eras who never had a chance to meet across the board. Recently Guid and Bratko [4] published an extensive computer analysis of World Chess Champions, aiming at such a comparison. It was based on the evaluation of the games played by the World Chess Champions in their championship matches. The idea was to determine the chess players' *quality of play* (regardless of the game score), which was evaluated with the help of computer analyses of individual *moves* made by each player. The winner according to the main criterion, where average deviations between evaluations of played moves and best-evaluated moves according to the computer were measured, was Jose Raul Capablanca, the 3rd World Champion, which to many came as a surprise (although Capablanca is widely accepted as an extremely talented and a very accurate player).

A version of that article was republished by a popular chess website, ChessBase.com [3], and various discussions took place at different blogs and forums across the internet, while the same website soon published some interesting responses by various readers from all over the world, including some by scientists [2]. A frequent comment by the readers could be summarised as: "A very interesting study,

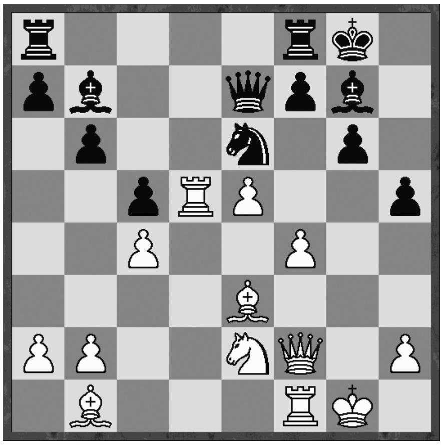
[1] Artificial Intelligence Laboratory, Faculty of Computer and Information Science, University of Ljubljana, Slovenia. Email: {matej.guid,bratko}@fri.uni-lj.si.

[2] Intelligent Systems Group, Department of Computer Science and Artificial Intelligence, University of the Basque Country. Email: aritz@si.ehu.es.

but it has a flaw in that program CRAFTY, whose rating is only about 2620, was used to analyse the performance of players stronger than this. For this reason the results cannot be useful". Some readers speculated further that the program will give better ranking to players that have a similar strength to the program itself. In more detail, the reservations by the readers included three main objections to the used methodology:

- the program used for analysis was too weak,
- the depth of the search performed by the program was too shallow[3],
- the number of analysed positions was too low (at least for some players).

In this paper we address these objections in order to determine how reliable CRAFTY (or any other fallible chess program) is as a tool for comparison of chess players, using the suggested methodology. In particular, we were interested in observing to what extent is the ranking of the players preserved at different depths of search. Our results show, possibly surprisingly (see Fig. 1), that at least for the players whose score differentiate enough from the others (as is the case for Capablanca and Kramnik on one side of the list, and Euwe and Steinitz on the other) the ranking remains preserved, even at very shallow search depths.



depth	best move	evaluation
2	Bxd5	-1.46
3	Bxd5	-1.44
4	Bxd5	-0.75
5	Bxd5	-1.00
6	Bxd5	-0.60
7	Bxd5	-0.76
8	Rad8	-0.26
9	Bxd5	-0.48
10	Rfe8	-0.14
11	Bxd5	-0.35
12	Nc7	-0.07

Fig. 1. Botvinnik-Tal, World Chess Championship match (game 17, position after white's 23rd move), Moscow 1961. In the diagram position, Tal played 23…Nc7 and later won the game. The table on the right shows CRAFTY's evaluations as results of different depths of search. As it is usual for chess programs, the evaluations vary considerably with depth. Based on this observation, a straightforward intuition suggests us that by searching to different depths, different rankings of the players would have been obtained. However, as we demonstrate in this paper, the intuition may be misguided in this case, and statistical smoothing prevails.

It is well known for a long time that strength of computer chess programs increases with search depth. Already in 1982, Ken Thompson [8] compared programs that searched to different search depths. His results show that searching to only one ply

[3] Search depth in the original study was limited to 12 plies (13 plies in the endgame) plus quiescence search.

deeper results in more than 200 rating points stronger performance of the program. Although later it was found that the gains in the strength diminish with additional search, they are nevertheless significant at search depths up to 20 plies [6]. The preservation of the rankings at different search depths would therefore suggest not only that the same rankings would have been obtained by searching deeper, but also that using a stronger chess program would not affect the results significantly, since the expected strength of CRAFTY at higher depths (e.g. at about 20 plies) are already comparable with the strength of the strongest chess programs, under ordinary tournament conditions at which their ratings are measured (see [7] for details).

We also studied how the scores and the rankings of the players would deviate if smaller subsets of positions were used for the analysis, and whether the number of positions available from world championship matches suffices for successful ranking of the World Champions.

2 Method

We used the same methodology as Guid and Bratko [4] did in their study. Games for the title of "World Chess Champion", where the fourteen classic World Champions contended for or were defending the title, were selected for analysis. Each position occurring in these games after move 12 was iteratively searched to depths 2 to 12 ply. Search to depth *d* here means *d* ply search extended with quiescence search to ensure stable static evaluations. The program recorded best-evaluated moves and their backed-up evaluations for each search depth from 2 to 12 plies (Fig. 2). As in the original study, moves where both the move made and the move suggested by the computer had an evaluation outside the interval [-2, 2], were discarded and not taken into account in the calculations. In such clearly won positions players are tempted to play a simple safe move instead of a stronger, but risky one. The only difference between this and the original study regarding the methodology, is in that the search was not extended to 13 plies in the endgame. Obviously the extended search was not necessary for the aim of our analysis: to obtain rankings of the players at the different depths of search.

The average differences between evaluations of moves that were played by the players and evaluations of best moves suggested by the computer were calculated for each player at each depth of the search. The results are presented in Fig. 3.

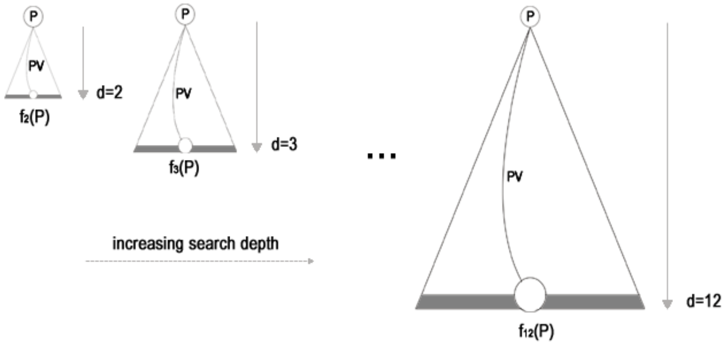


Fig. 2. In each position, we performed searches to depths from 2 to 12 plies extended with quiescence search to ensure stable static evaluations. Backed-up evaluations of each of these searches were used for analysis.

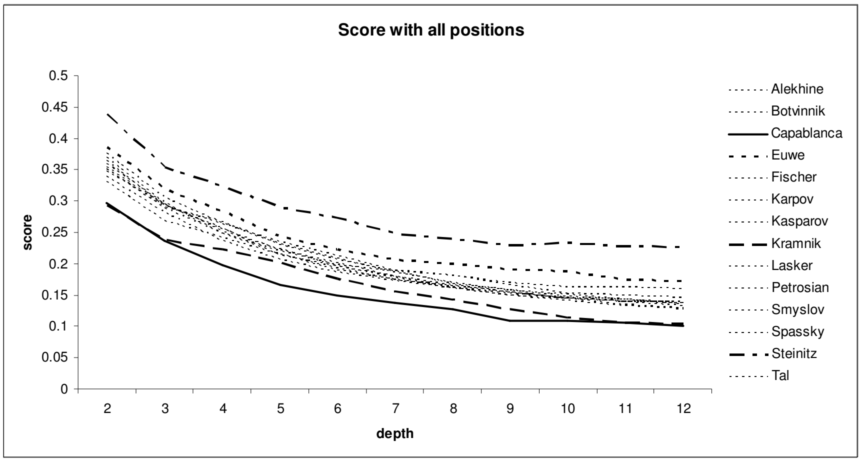


Fig. 3. Average scores (deviations between the evaluations of played moves and best-evaluated moves according to the computer) of each player at different depths of search. The players whose scores clearly deviate from the rest are Capablanca, Kramnik (in positive sense) and Euwe, Steinitz (in negative sense).[4]

The results clearly demonstrate that although the deviations tend to decrease with increasing search depth, the rankings of the players are nevertheless preserved, at least for the players whose scores differ enough from the others (see Fig. 4). It is particularly impressive that even trivial search to depth of two or three ply does rather good job in terms of the ranking of the players.

[4] The lines of the players whose results clearly deviate from the rest are highlighted. The same holds for figures 4, 6, and 8.

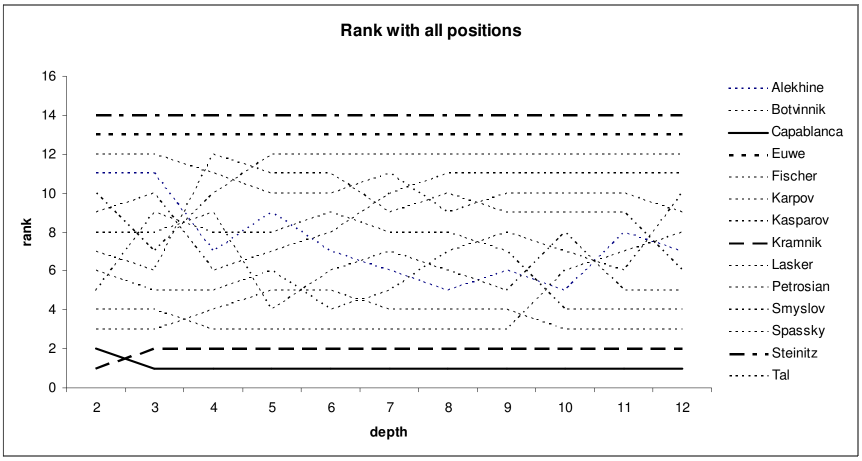


Fig. 4. Ranking of the players at different search depths.

In order to check the reliability of the program as a tool for ranking chess players, it was our goal to determine:

– the stability of the obtained rankings in different subsets of analysed positions,
– the stability of the rankings with increasing search depth.

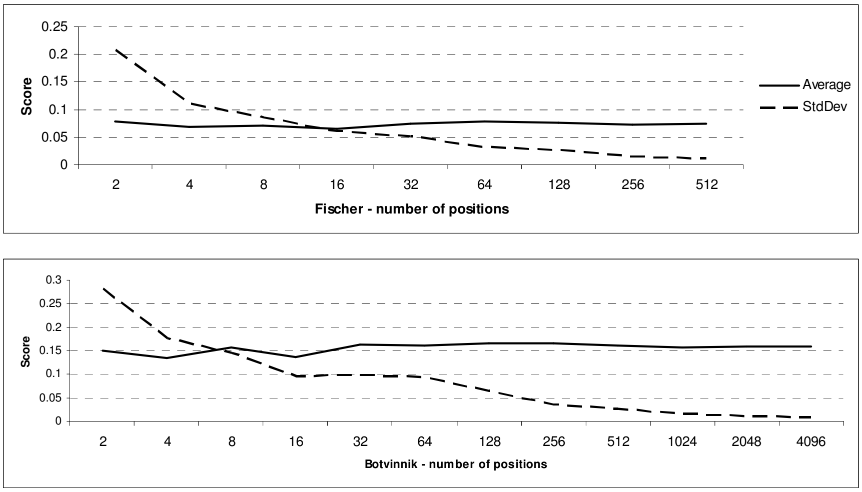


Fig. 5. Average scores of each player were computed for 1000 subsets of different sizes. The graph represents the results for players Fischer and Botvinnik, for subsets consisting of evaluations resulting from search to depth 12.

For each player, 100 subsets from the original dataset were generated by randomly choosing 500 positions (without replacement) from their games. The number of available positions varies for different players, since they were involved in a different number of matches. About 600 positions only were available for Fischer, while both for Botvinnik and Karpov this number is higher than 5100 at each depth. The exact number for each player slightly varies from depth to depth, due to the constraints of the methodology: positions where both the move made and the move suggested by the computer had an evaluation outside the interval [-2, 2] had to be discarded at each depth. Experiments with subsets of different sizes suggest that the size of 500 already seems to be sufficient for reliable results (Fig. 5).

We observed variability of scores and rankings, obtained from each subset, for each player and at each search depth. The results are presented in the next section.

3 Results

The results presented in this section were obtained on 100 subsets of the original dataset, as described in the previous section.

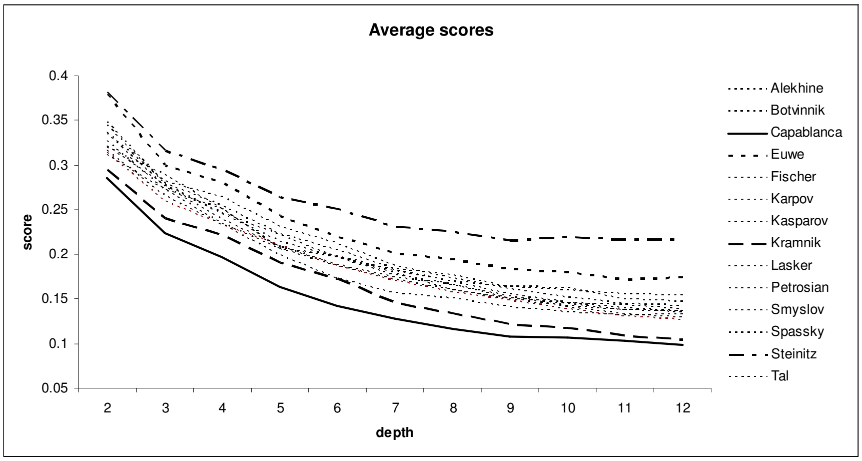


Fig. 6. Average scores of each player at different search depths.

Fig. 6 represents average scores of the players across all the subsets, at each search depth from 2 to 12. The obvious similarity to the graph in Fig. 3 confirms that the results obtained on the whole dataset were not coincidental. This conclusion was confirmed by observing average scores of the players across all depths for each subset separately: Capablanca had the best such score in 96% of all the subsets.

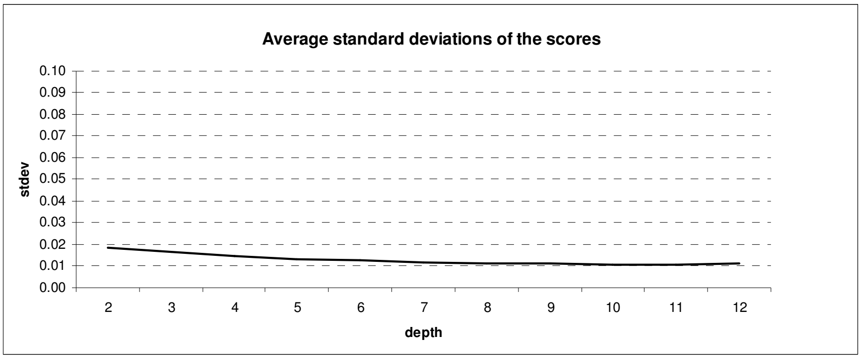


Fig. 7. Average standard deviations of the scores of the players. The scale is adjusted for easier comparison with the graph in Fig. 6.

The average standard deviations of the players' scores show that they are slightly less variable at higher depths. Anyway, they could be considered practically constant at depths higher than 7 (Fig. 7). All the standard deviations are quite low, considering the average difference between players whose score differ significantly.

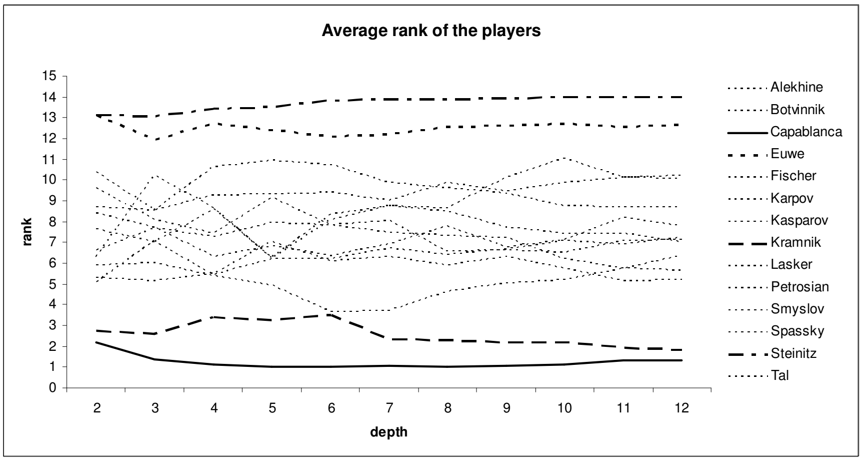


Fig. 8. Average rank of the players.

Fig. 8 (similar to Fig. 4) shows that the rankings preserve for Capablanca, Kramnik, Euwe and Steinitz, whose scores differ significantly from the rest of the players.

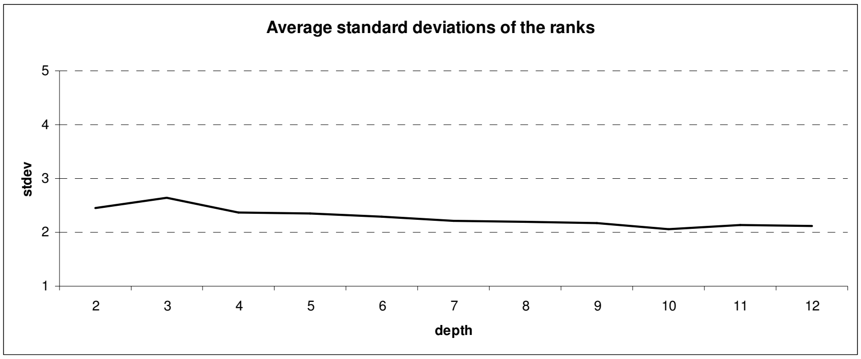


Fig. 9. Average standard deviations of the players' ranks (obtained in 100 subsets).

The average standard deviations of the players' ranks (obtained in 100 subsets) only slightly increase with increasing search depth and are practically equal for most of the depths (Fig. 9).

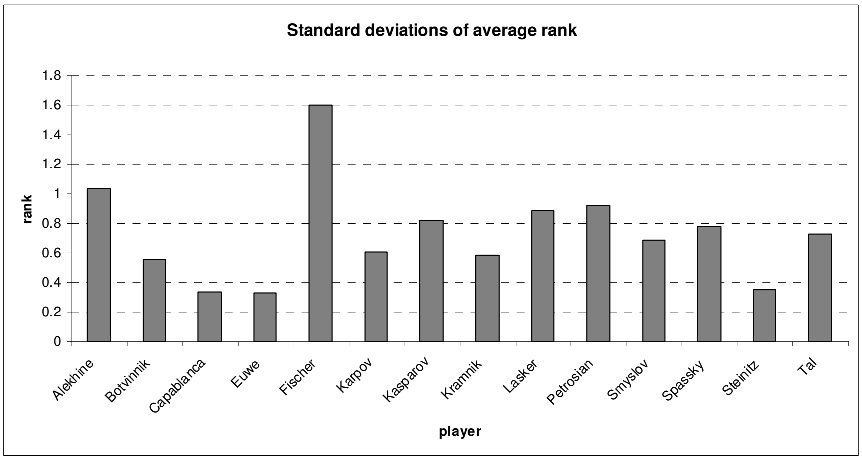


Fig. 10. Standard deviations of the average ranks for each player across all depths.

The graph of standard deviations of the average ranks from different depths for each player separately (Fig. 10) confirms that the rankings of most of the players on average preserve well across different depths of search.

4 A Simple Probabilistic Model of Ranking by Imperfect Referee

Here we present a simple mathematical explanation of why an imperfect evaluator may be quite sufficient to correctly rank the candidates. The following simple model was designed to show the following:

- To obtain a sensible ranking of players, it is not necessary to use a computer that is stronger than the players themselves. There are good chances to obtain a sensible ranking even using a computer that is weaker than the players.
- The (fallible) computer will not exhibit preference for players of similar strength to the computer.

Let there be three players and let us assume that it is agreed what is the best move in every position. Player *A* plays the best move in 90% of positions, player *B* in 80%, and player *C* in 70%. Assume that we do not know these percentages, so we use a computer program to estimate the players' performance. Say the program available for the analysis only plays the best move in 70% of the positions. In addition to the best move in each position, let there be 10 other moves that are inferior to the best move, but the players occasionally make mistakes and play one of these moves instead of the best move. For simplicity we take that each of these moves is equally likely to be chosen by mistake by a player. Therefore player *A*, who plays the best move 90% of the time, will distribute the remaining 10% equally among these 10 moves, giving 1% chance to each of them. Similarly, player *B* will choose any of the inferior moves in 2% of the cases, etc. We also assume that mistakes by all the players, including the computer, are probabilistically independent.

In what situations will the computer, in its imperfect judgement, credit a player for the "best" move? There are two possibilities:

1. The player plays the best move, and the computer also believes that this is the best move.
2. The player makes an inferior move, and the computer also confuses this *same* inferior move for the best.

By simple probabilistic reasoning we can now work out the computer's approximations of the players' performance based on the computer's analysis of a large number of positions. By using (1) we could determine that the computer will report the estimated percentages of correct moves as follows: player *A*: 63.3%, player *B*: 56.6%, and player *C*: 49.9%. These values are quite a bit off the true percentages, but they nevertheless preserve the correct ranking of the players. The example also illustrates that the computer did not particularly favour player *C*, although that player is of similar strength as the computer.

$$P' = P \cdot P_C + (1 - P) \cdot (1 - P_C) / N \qquad \mathbf{(1)}$$

P = probability of the player making the best move
P_C = probability of the computer making the best move
P' = computer's estimate of player's accuracy P

N = number of inferior moves in a position

The simple example above does not exactly correspond to our method which also takes into account the cost of mistakes. But it helps to bring home the point that for sensible analysis we do not necessarily need computers stronger than human players.

5 A More Sophisticated Mathematical Explanation

How come the rankings of the players, as the results demonstrate, preserve rather well, despite the big differences in evaluations across different search depths? In the sequel we attempt to provide an explanation for this phenomenon.

Suppose we have an estimator A that measures the goodness of an individual *M* in a concrete task, by assigning this individual a score (*S*), based on some examples. The estimator assigns different scores to the respective individuals and therefore has a variance associated:

$$Var_M^A = E\left(S_M^A - E\left(S_M^A\right)\right)^2 \tag{2}$$

The estimator gives an approximation (S_M^A) of the real score (S_M) of the individual, which results in a bias:

$$Bias_M^A = E\left(S_M^A - S_M\right) \tag{3}$$

The probability of an error in comparison of two individuals, *M* and *N*, using the estimator *A*, only depends on the bias and the variance. Given two different estimators, *A* and *B*, if their scores are equally biased towards each individual ($Bias_M^A$ = $Bias_N^A$ and $Bias_M^B$ = $Bias_N^B$) and variances of the scores of both estimators are equal for each respective individual (Var_M^A = Var_M^B and Var_N^A = Var_N^B), then both estimators have the same probability of committing an error (Fig. 11).

This phenomenon is commonly known in the machine-learning community and has been frequently used, e.g., in studies of performances of estimators for comparing supervised classification algorithms [1, 5]. In the sequel we analyse what happens in comparisons in the domain of chess when estimators based on CRAFTY at different search depths are used, as has been done in the present paper.

In our study the subscript of S_M^A refers to a player and the superscript to a depth of search. The real score S_M could not be determined, but since it is commonly known that in chess the deeper search results in better heuristic evaluations (on average), for each player the average score at depth 12, obtained from all available positions of each respective player, served as the best possible approximation of that score. The biases and the variances of each player were observed at each depth up to 11, once again using the 100 subsets, described in Section 2.

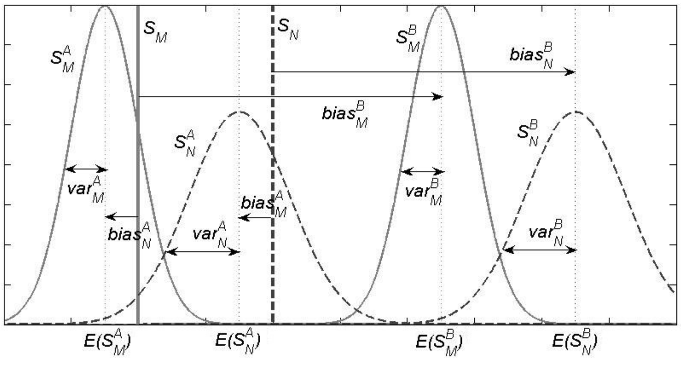


Fig. 11. Although estimators *A* and *B* give different approximations of the real scores of individuals *M* and *N* (S_M and S_N), and *A* approximates the real scores more closely, since their scores are equally biased towards each individual ($Bias_M^A = Bias_N^A$ and $Bias_M^B = Bias_N^B$) and variances of the scores of both estimators are equal for each respective individual ($Var_M^A = Var_M^B$ and $Var_N^A = Var_N^B$), they are both equally suitable for comparison of *M* and *N*.

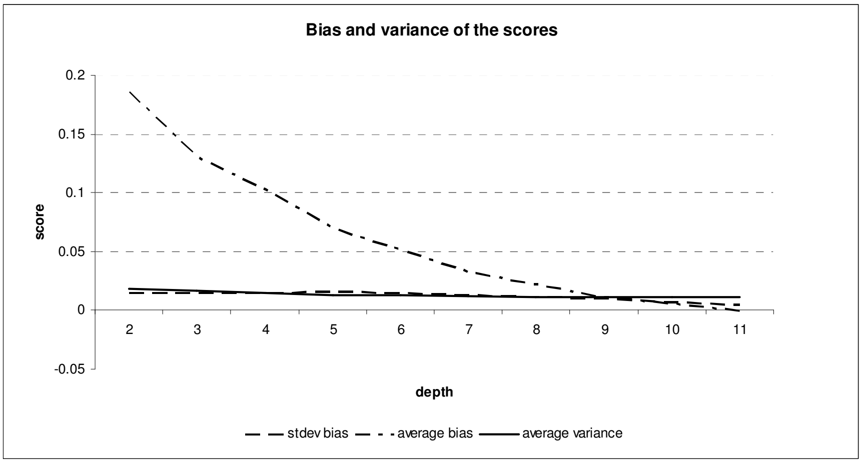


Fig. 12. Average biases, standard deviations of them, and standard deviations of the scores.

The results are presented in Fig. 12. The standard deviation of the bias over all players is very low at each search depth, which suggests that $Bias_M^A$ is approximately equal for all the players *M*. The program did not show any particular bias at any depth towards Capablanca nor towards any other player. Moreover, the standard deviation is practically the same at all levels of search with only a slight tendency to decrease with increasing search depth. Standard deviations of the scores are also very low at all depths, from which we could assume that $Var_M^A = Var_M^B$ also holds. For better visualisation we only present the mean variance, which as well shows only a slight tendency to decrease with depth. To summarise, taking into account both of these facts, we can conclude that the probability of an error of comparisons performed by CRAFTY at different levels of search is practically the same, and only slightly diminishes with increasing search depth.

6 Conclusion

In this paper we analysed how trustworthy are the rankings of chess champions, produced by computer analysis using the program CRAFTY [4]. In particular, our study was focused around frequently raised reservations expressed in readers' feedback: (1) the chess program used for the analysis was too weak, (2) the depth of the search performed by the program was too shallow, and (3) the number of analysed positions was too low (at least for some players).

The results show that, at least for the two highest ranked and the two lowest ranked players, the rankings are surprisingly stable over a large interval of search depths, and over a large variation of sample positions. It is particularly surprising that even extremely shallow search of just two or three ply enable reasonable rankings. Indirectly, these results also suggest that using other, stronger chess programs would be likely to result in similar rankings of the players.

References

[1] Alpaydin E. Combined 5 x 2 cv F Test for Comparing Supervised Classification Learning Algorithms. *Neural Computation*, Vol.11, No. 8, pp. 1885-1892, 1999.

[2] *Chessbase.com*: Computer analysis of world champions. http://www.chessbase.com/newsdetail.asp?newsid=3465

[3] *Chessbase.com*: Computers choose: who was the strongest player? http://www.chessbase.com/newsdetail.asp?newsid=3455

[4] Guid M. and Bratko I. Computer analysis of world chess champions. *ICGA Journal*, Vol. 29, No. 2, pp. 65-73, 2006.

[5] Kohavi R. A study of cross-validation and bootstrap for accuracy estimation and model selection. *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (ed. C. S. Mellish), Morgan Kaufmann Publishers, Inc., pp. 1137-1143, 1995.

[6] Steenhuisen J. R. New results in deep-search behaviour. *ICGA Journal*, Vol. 28, No. 4, pp. 203-213, 2005.

[7] *The SSDF Rating List*: http://web.telia.com/~u85924109/ssdf/list.htm

[8] Thompson, K. Computer chess strength. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), Pergamon Press, pp. 55-56, 1982.

Moving in the Dark: Progress through Uncertainty in Kriegspiel

Andrea Bolognesi and Paolo Ciancarini

Dipartimento di Scienze dell'Informazione
University of Bologna, Italy

Abstract. Kriegspiel is a wargame based on the rules of Chess. A player only knows the position of his own pieces, while his opponent's pieces are "in the dark", ie. they are invisible. A Kriegspiel player has to guess the state of the game and progress to checkmate being in the dark about the history of moves (but he can exploit the messages of a referee). Thus, computer playing of Kriegspiel is difficult, because a program has to progress in the game even with scarce or no information on its opponent's position. It is especially interesting to study the progress of computer play of simple Kriegspiel endings. We show how we tested a program able to play simple Kriegspiel endings to see how it progresses through uncertainty.

1 Introduction

Kriegspiel is a Chess variant similar to wargames. Each player uses a normal chessboard with normal pieces and normal rules, except that he cannot see his opponent's pieces. Both players are not informed of their opponent's moves. Each move is tried "in the dark", ie. knowing nothing about the position and strategy of the opponent.

Since Kriegspiel is based on Chess, a normal Chess program can be easily adapted to play Kriegspiel, for instance trying random moves until the referee accepts one. If we want to have a playing quality better than random, however, a special problem has to be addressed. Most computer chess evaluation functions compute a score evaluating both armies, whose position is well known, and then the search procedure progresses maximizing or minimizing the difference of score assigned to each army. In Kriegspiel this optimization is not possible, so progress (namely improving the program's army position with respect to the position of the adversary) becomes a problem. A player has to choose a "good" move being in the dark about the position of the enemy army.

In order to build a complete program able to play a good Kriegspiel game, the study of simple endings is useful because these endings are quite common in the practice of the game between humans. There is also a game-theoretic interest: in fact, a number of papers discuss abstract, rule based procedures suitable to solve the simplest endings from any initial position. A first study on the the ♔♖♚ ending was published by Boyce, who proposed a complete procedure to solve it [1]. Also two Italian researchers studied this ending, more or less at the same time than Boyce's [2] Then Ferguson analysed the endings ♔♗♘♚ [3]

and ♔♗♗♚ [4]. A rule-based program to play the ♔♙♚ ending according to the principle of Bound Rationality was discussed in [5].

In this article we study and evaluate instead a search-based algorithm first exposed in [6] and generalized in [7] . It explores a game tree made of nodes which are metapositions [8], and uses an evaluation function in order to implement a progress heuristic.

We will deal with the basic endgames, i.e. those where Black has left the King only. This paper completes the work described in our papers [6,7]: there we discussed some partial results on the most common Kriegspiel endings, that we here consolidate and generalize. More precisely, in [6] we developed a basic program for the classic ending ♔♖♚; in [7] we extended our approach, developing a more general evaluation function useful also for other endings.

The goal of this paper is to study how our approach is effective, namely we aim at showing that our algorithm progresses even if it ”moves in the dark”. We initially compare our algorithm with an abstract algorithm proposed by Boyce [1]. The Boyce algorithm is rule-based and in our knowledge has been never implemented before. Our algorithm instead is search-based: our goal is to compare the two different algorithms from a practical, agonistic viewpoint. In fact, we have recently developed a complete playing program named Darkboard [9]: when we let it to play on the Internet Chess Club (ICC), human Kriegspiel players are very clever in exploiting its weaknesses in dealing with simple endings.

This paper has the following structure. In section 2 we evaluate our approach comparing it with the algorithm proposed by Boyce. In Section 3 we discuss the completeness of our algorithm. In Section 4 we draw our conclusions.

2 Tests and comparisons

In order to evaluate the quality of the algorithm described in[7], we have implemented another, different, rule based program which plays the procedure proposed in [1] to win the Rook ending.

2.1 The Boyce algorithm

Boyce showed a way to force checkmate by considering positions where both Kings are in the same quadrant of the board as seen from the rook, or where the Black King is restricted to one or two quadrants of the board.

The procedure applies when

1. both kings are in the same quadrant as designed by the rook; see Fig. 1;
2. the black king cannot exit from the quadrant;
3. the white rook is safe.

Basically, the algorithm first ensures that the rook is safe from capture. Next White plays to a position where all the possible squares for the Black king are in a rectangle where one corner is at the rook. White will put its king in that

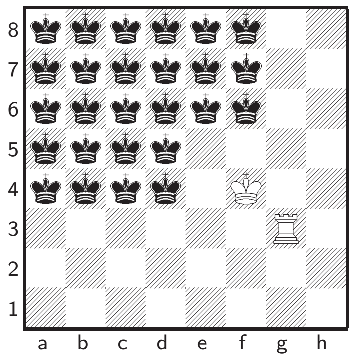


Fig. 1. Initial position of the Boyce's procedure

rectangle to keep the Black king away from its rook. White then forces the Black king back until it can occupy only those squares on a single edge. The final phase to checkmate is then fairly simple.

We have implemented a program which uses a search algorithm and a special evaluation function with the aim to obtain an initial position similar to that shown in Fig. 1. Then we apply the Boyce rules, and count how many moves are necessary in order to give mate.

2.2 Our algorithm

Our search-based algorithm has been presented in [7]. Here we summarize only the main ideas in its evaluation function. The function includes six different euristics.

1. it avoids jeopardizing the Rook;
2. it brings the two Kings closer;
3. it reduces the size of the quadrant where the Black king should be found;
4. it avoids the Black king to go between White rook and White king;
5. it keeps the White pieces close to each other;
6. it pushes the Black King toward the corner of the board.

These features are evaluated numerically and added to obtain the value for a given metaposition: a search program then exploits the evaluation function to visit and minimax a tree of metapositions [7].

2.3 Comparing the two programs

Figure 2 shows a graph which depicts the result of all the 28000 matches which can occur considering all the possible initial metapositions for the rook ending from the White's point of view, starting with greatest uncertainty, that is starting from metapositions where each square not controlled by White may contain a Black king. The number of matches won is on the ordinate and the number

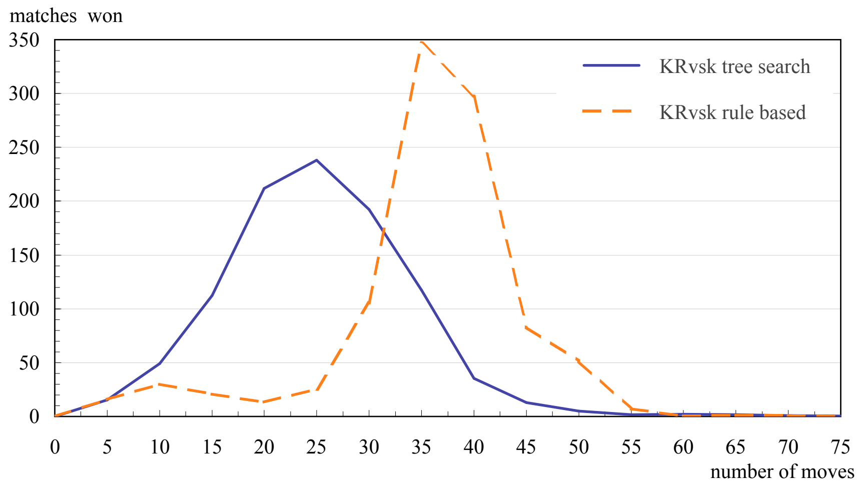


Fig. 2. Comparison of the rule-based program with the search-based program

of moves needed to win each match is on the abscissa. The graphic shows the distribution of the matches won normalized to 1000.

The rule based program spends the first 25 moves looking for one of the initial positions; when it reaches one of these positions the checkmate procedure is used and the program wins very quickly. However, the average of moves needed is around 35. Our program based entirely on the search of game tree wins with a better average, around 25 moves.

This is due to the fact that the program analyzes from the beginning each position trying to progress to checkmate. On the other hand, the rule-based program is faster in deciding the move to choose, with respect to the tree-searching program. In fact, the rule-based program has a constant running time, whereas the second one has a running time exponential on the game tree depth.

We remark, however, that from a practical viewpoint the Boyce approach is useless because on the ICC Kriegspiel is played with the normal 50-moves draw rules derived from chess.

2.4 Evaluating the search algorithm with other endings

Figure 3 shows the results obtained with our search-based algorithm when analyzing some different basic endings. We performed a test choosing random metapositions with greatest uncertainty for ♔♕♚, ♔♗♗♚, and ♔♗♘♚ endings; then we normalized the results to 1000 and we merged them to produce the ♔♖♚ figure.

In figure 3 we see that the program wins the ♔♕♚ ending quicker than the ♔♖♚ ending. This result was expected, because the queen is more powerful

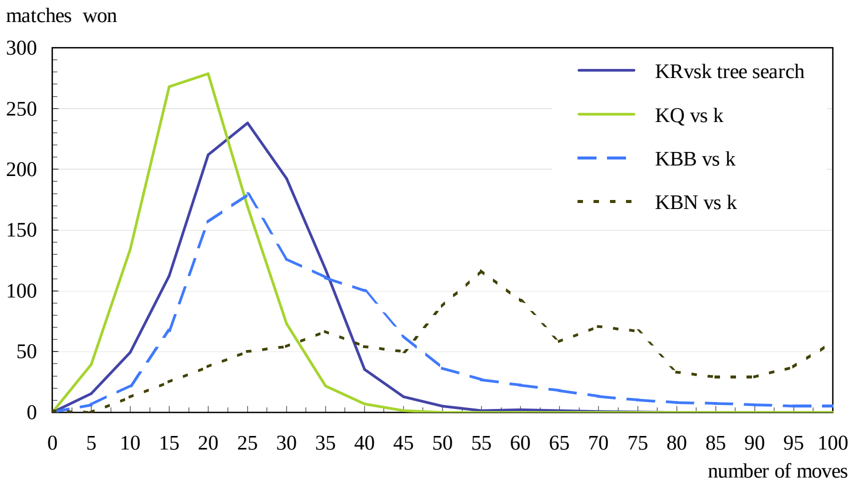


Fig. 3. Comparing the behavior of the search-based algorithm on different endings

than the rook: the queen controls more space so metapositions have a lesser degree of uncertainty.

The case ♔♗♗♚ is instead more difficult with respect to ♔♖♚. In fact, the former is won on average in a larger number of moves: sometimes our program needs more than 100 moves.

Finally, we see that the behavior of our program in the ♔♗♘♚ ending is not good at all. The program often spends more than 100 moves to win and the distribution of victories does not converge to zero, meaning that sometimes it takes an unbound number of moves to win. We conclude that in this ending our program is not really able to progress.

2.5 Progress through Uncertainty

An effective way to analyze the progress toward victory consists in considering how the value of White's reference board changes after playing each pseudomove. The reference board is the metaposition which describes all positions were the opponent King can be, compatibly with the past history of the game.

Figure 4 shows the trend of evaluations assigned to each reference board reached during a whole match for the ♔♕♚ ending. The number of attempts needed during the game is shown on the abscissa, while the grades assigned by the evaluation function are on the ordinate.

We see that, at each step, the value of metapositions increases. From White's point of view, this means that the state of the game is progressing and this is actually a good approximation for the real situation.

We have performed the same test for the case of ♔♗♘♚ ending, whose result is depicted in Figure 5. Here the progress is not satisfactory for White, in

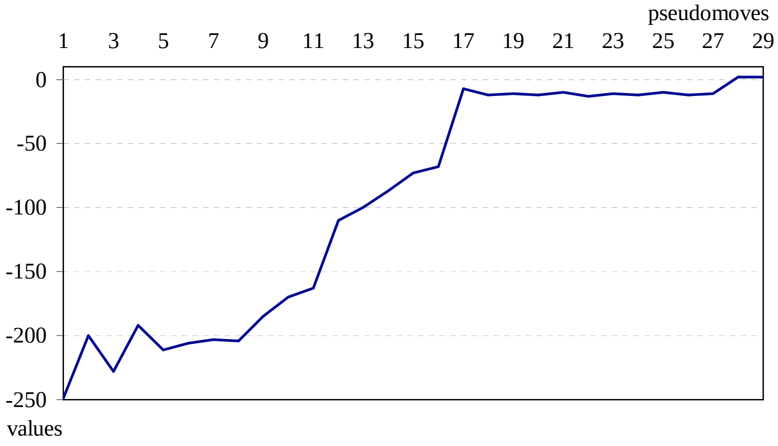


Fig. 4. Trend of evaluations assigned to metapositions crossed during ♔♕♚ ending

fact he does not improve the state of the game at each step. The graph shows how the evaluations of the reference board change during a match which ends with the win of White: the value of metapositions does not increase at each pseudomove, but at some lucky situation for White. Thus the program basically wins this game by chance, that is by exploiting either some lucky metapositions or its opponent's errors.

We conclude that our program is able to progress to victory when we deal with pieces able to divide the board in separate areas, which can then be reduced to trap the Black King; whereas when we have a piece which does not offer this feature, like the Knight, the behavior of the program is not fully satisfactory.

3 Optimality

In this section we deal with the issue of the optimality of our approach. We will show that our program is not able to win against an opponent using an oracle. If it won against an omniscient opponent it would be able to give optimal solutions, due to the search algorithm properties. We will point out that problems arise when the possibility of an illegal answer is considered into the tree of moves.

We remark that, according to its author, the algorithm given in [1] (or the (different) one given in [2]) can win against an omniscient opponent. However, these abstract algorithms do not always chose the optimal [1] move for each position the player may be in. Alas, these solutions follow a strategy fixed in advance that let the player to win even against an opponent using an oracle.

We showed in [7] that with a search on the tree of moves our program can guarantee the optimality if a checkmate position is among the visited nodes. In

[1] i.e. the move which leads to victory with the minimum number of moves

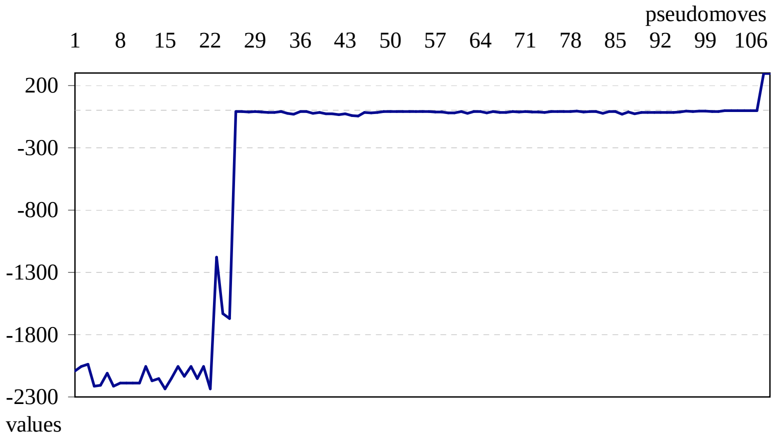


Fig. 5. Trend of evaluations assigned to metapositions crossed during ♔♗♘♚ ending

all the other cases what we can say is that the move chosen by the program depends on the evaluation function which judges the positions. In these cases we cannot claim optimality.

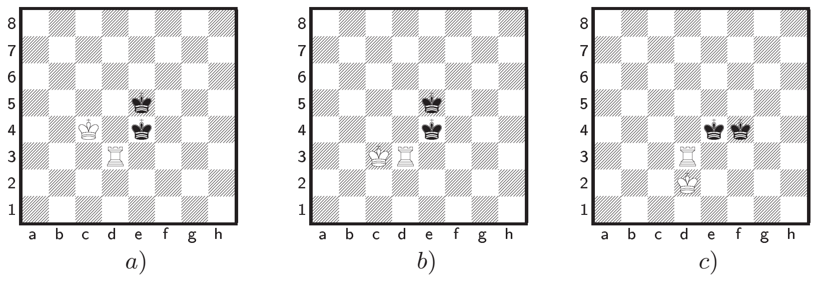


Fig. 6. Difficult positions against an omniscient opponent

We will see in this section that the program with the evaluation function proposed in [7] does not always win against an omniscient opponent.

If the program ends up in the position *a*) depicted in figure 6 no checkmate state is found during the search on the tree of moves, so the program entrusts the evaluation function with the task to judge for sub-optimal positions. Since the evaluation function for the ♔♖♚ ending tries to reduce the uncertainty about the black King by decreasing the number of black kings on his reference board, it tries to push the white King inside the quadrant controlled by the rook. ♔d5 and ♔d4 are illegal moves, so the program plays ♔c3: the program plays subsequently ♔d2, then ♔e3, trying to pass through the row controlled by the rook.

If ♔e3 is a legal move, then the referee remains silent and White can proceed, but if the referee says 'illegal', then the game reaches the position *c*) depicted in Figure 6. In such a case the program acts as if it would play for time: it chooses ♔e2 or ♔c3. This behavior depends on a random choice among the moves with same value. So if it plays ♔c3 then the game comes up again as in the initial position where we started, otherwise if it plays ♔e2, then at the subsequent turn it will try ♔e3 again.

We note that the program's behavior is not optimal: it entrusts its choices to the chance of having a silent referee after its move. In other words, if White plays ♔e3 and the referee is silent then the program progresses; actually it decreases the uncertainty about the black king's position. If it plays ♔e3 and the referee's answer is "illegal", then it does not progress.

If Black has no hint about White's moves, then a good strategy consists in centralizing his king to the middle of the chessboard. For instance, the black king could move from e5 onto e4 and viceversa, as highlighted in miniature *d*) in figure 7 on the left. In this case, the program with the evaluation function given in [7] progresses and its choices lead him to win the game.

On the contrary, if Black gets an oracle to help him, it can halt White's progress. In this case, White faces an omniscient opponent and the program fails by going forwards and backwards from position *b*) in Figure 6 to position *c*) in Figure 6.

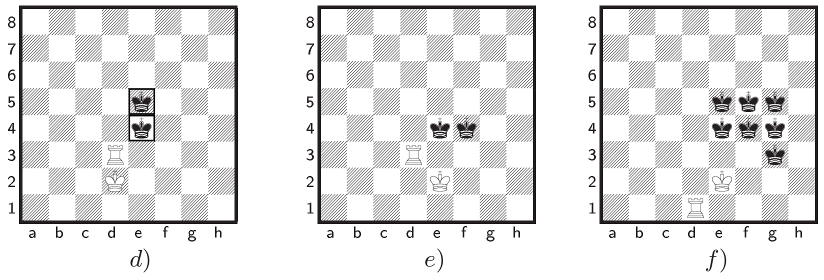
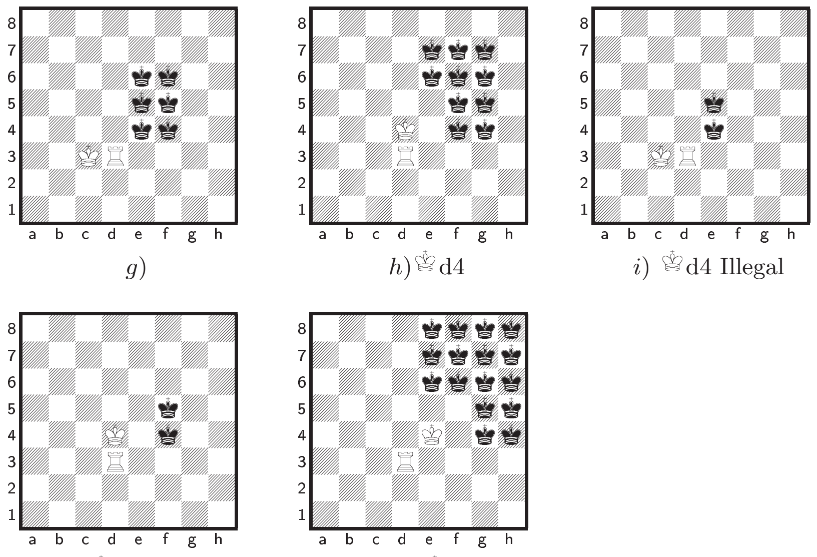


Fig. 7. Difficult positions against an omniscient opponent

From [1] we know that a good move for White consists in playing ♖d1, when the king is in e2 square (position *e*) in Figure 7). In fact this move allows White having his king inside the quadrant controlled by his rook. With this move White reaches a positions from which he can progresses.

Writing an evaluation function which considers the position *f*) of figure 7 better then the position *b*) in figure 6 is quite a complex matter.

This problem can be expressed observing that from position *a*) in Figure 6, moving ♔c3 White reaches the position *g*) in Figure 8, then the move ♔d4 may receive two different answers from the referee: silent which leads to position *h*) in Figure 8, or illegal which leads to position *i*) in Figure 8. The worst answer is the silent one[2], so the value for ♔d4 will be considered as the value of the position reached with a 'silent' answer.



g)   *h*)♔d4   *i*) ♔d4 Illegal

l) ♔e4 Illegal   *m*) ♔e4

Fig. 8. Difficult positions against an omniscient opponent

Again, starting from position *h*) in Figure 8 if White plays ♔e4 he may receive two kind of answer: silent or illegal. Also in this case the illegal answer is better because offers more information.

As a result, during the search each time the program analyzes moves of the king in situations similar to the position *g*) or the position *h*) of Figure 8, the illegal answer is not taken into account.

In Figure 9 is proposed an example of game tree. Starting from the position depicted on the root of the tree we have only represented three moves that White can choose.

We want to focus on the reasons that lead the program to prefer ♔c3 rather than ♔e2. Each time that a move can be illegal, this kind of answer leads to a node in the tree that is better valued with respect to the node resulting from a silent referee. Since the program considers the value of a move equal to the value of the worst position reached considering all the possible referee's answers, if a move may be illegal or silent then the worst one is the silent one.

Figure 9 shows that the leaf of the leftmost branch represents a better position than the leaf of the rightmost branch. In the middle there is a branch

[2] In order to progress, the number of black kings in the reference board computed after each try should decrease: the best answer from the referee is 'illegal', because we can infer that the opponent king must be close to our king.

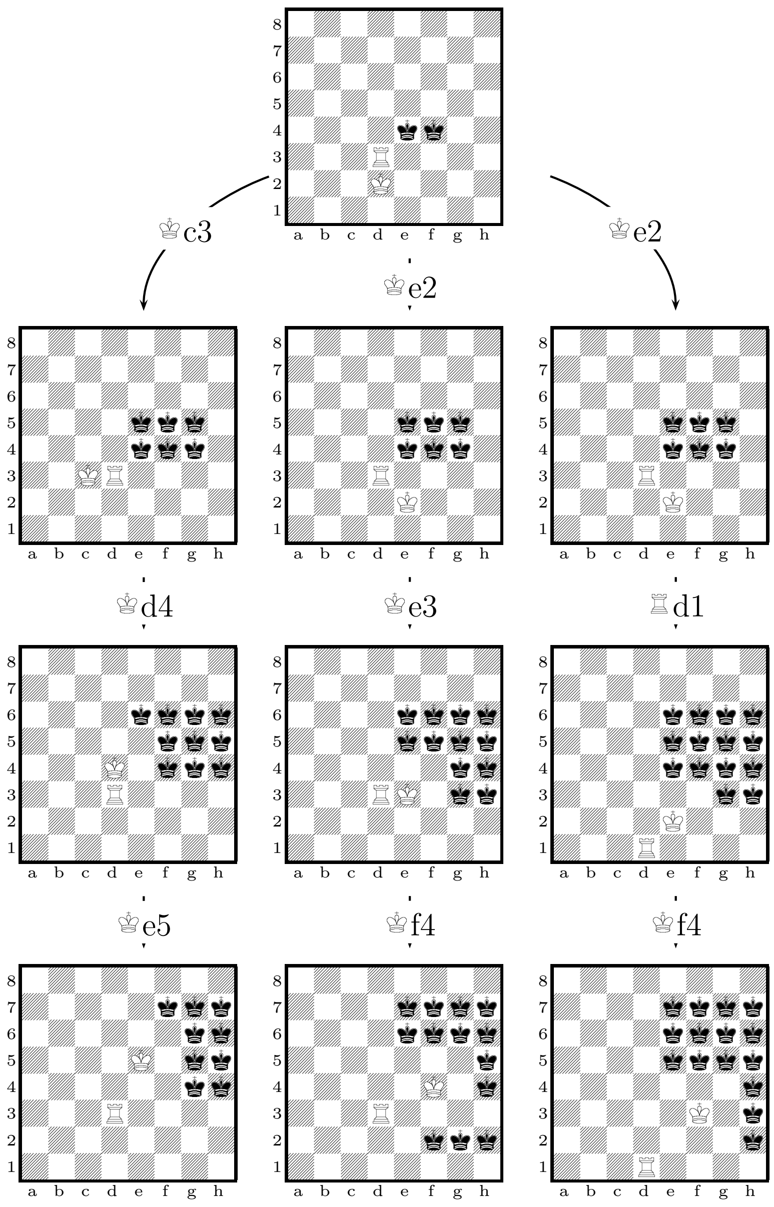


Fig. 9. Example of tree of metapositions

concerning the ♔e2;♔e3 moves which lead to bad positions. The problem is due to a program blindness: it acts as if it was not distinguishing a potentially illegal branch from a certainly legal one.

An attempt to solve this problem consists in decreasing the value of illegal moves. If we do it too heavily we run the risk of letting the program to play only moves that are legal. In order to overcome this problem, we can try to discard from the search on the tree of moves the positions previously reached during the game. This is a trick that can avoid infinite loops, but it does not lead us to formulate an optimal solution to the problem.

4 Conclusions

The work done so far on a program which plays Kriegspiel endings exploiting the search on a tree of moves lets us to obtain optimal result for endings in which the checkmate is technically achievable in few moves. In all the other cases we delegate the judgement on moves to an evaluation function, that we proposed in [6] and we have generalized in [7].

The approximation due to this kind of evaluation lets the program to play reasonably against a fair opponent, which does not use an oracle. However the program plays not optimally in positions with larger uncertainty or against an omniscient opponent.

References

1. Boyce, J.: A Kriegspiel Endgame. In Klarner, D., ed.: The Mathematical Gardner. Prindle, Weber & Smith (1981) 28–36
2. Leoncini, M., Magari, R.: Manuale di Scacchi Eterodossi. Tipografia Senese, Siena (1980)
3. Ferguson, T.: Mate with Bishop and Knight in Kriegspiel. Theoretical Computer Science **96** (1992) 389–403
4. Ferguson, T.: Mate with two Bishops in Kriegspiel. Technical report, UCLA (1995)
5. Ciancarini, P., DallaLibera, F., Maran, F.: Decision Making under Uncertainty: A Rational Approach to Kriegspiel. In van den Herik, J., Uiterwijk, J., eds.: Advances in Computer Chess 8, Univ. of Rulimburg (1997) 277–298
6. Bolognesi, A., Ciancarini, P.: Computer Programming of Kriegspiel Endings: the case of KR vs K. In van den Herik, J., Iida, H., Heinz, E., eds.: Advances in Computer Games 10, Kluwer (2003) 325–342
7. Bolognesi, A., Ciancarini, P.: Searching over Metapositions in Kriegspiel. In van den Herik, J., Bjornsson, Y., Netanyahu, N., eds.: Revised papers from 4th Int. Conf. on Computer and Games. Number 3846 in LNCS, Springer (2006) 246–261
8. Sakuta, M., Iida, H.: Solving Kriegspiel-like Problems: Exploiting a Transposition Table. ICCA Journal **23** (2000) 218–229
9. Ciancarini, P., Favini, G.: Representing Kriegspiel States with Metapositions. In: Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI 07), India (2007) 2450–2455

Extracting Important Features by Analyzing Game Records in Shogi

Kosuke Tosaka[1], Asuka Takeuchi[2], Shunsuke Soeda[2], and Hitoshi Matsubara[2]

[1] NTT Data Technology
tosakak@nttdtec.co.jp
[2] Future University-Hakodate , 116-2Kamedanakano-cho, Hakodate, Hokkaido, Japan 041-8655
{g2107019, shnsk, matsubar}@fun.ac.jp

Abstract. In this paper, we propose a method to extract differences of two groups of players, by statistically analyzing their game records. We conducted a discriminant analysis on the game records, choosing the group of players as the explanatory variable, and the combination of simple factors based on the rules of the game as the features. Combining all factors is unrealistic, as they would create an enormous number of the features, most of them unnecessary for the analysis. Thus, we have used a stepwise method to select meaningful variables, and then made an analysis based on the combination of the selected variables. By reducing number of the variables this way, we were able to achieve high discriminant rates.

1 Introduction

There has been a steady progress in programs playing Shogi, reaching the level of top-level human amateur players [7]. This has been due to several improvements, including faster hardware, sophisticated search algorithms, and better evaluation functions. The design of the evaluation functions is especially important, as it influences not only the strength, but also the playing style of the program. Currently, most evaluation functions use by the top-level programs are hand tuned, based on the knowledge of the programmer.

The evaluation functions are made by combinations of *features*, each representing some aspects of the state of a game position. Numerous features have been proposed to be used in the game playing programs, for example the number of the pieces each player owns and the relative positions of two pieces on the board. The playing style of a program largely depends on the features chosen to be used in the evaluation function. However, it is not well known how each feature influences the strength and the playing style of the programs.

In this paper, we propose a method to find out how each feature influences the strength and the playing style, by statistically analyzing game records. The game records are divided into two groups, and a discriminant analysis is performed on the two groups with the features as the explanatory variable; which differentiate the two groups are the features of the different playing style and the strength.

We conducted two experiments, comparing different types of the players. In the first experiment, we compared the game records of Shogi Grandmaster Habu and the other top-level Shogi professionals. In the second experiment, we compared the game records of the games between human players and the games between Shogi playing programs. In the first experiment, we were able to select features that represent the differences between the two groups. In the second experiment, more fundamental features such as the piece frequently used and the number of check moves, were enough to discriminate the two groups.

2 Related Work

Game records are used in the research of games in several ways. The realization probability search algorithm uses the game records to train the parameters to control the search [13]. Game records are also used to train the parameters for evaluation functions [10, 5, 2]. Some opening book databases are generated from game records [2].

Realization probability search algorithm (RP search) uses a value called realization probability (RP) to control how deep each sequence of moves should be searched in the game tree [13]. RP is a value associated to a move, based on the characteristic of the move (e.g. capturing move, check move, and the type of piece moved). Instead of using depth as the threshold, in RP-search the sum of RP is used as the threshold of the depth of each search path. The Shogi program "Gekisashi" uses the RP-search algorithm, learning the RP from the game records played by the Shogi Grandmaster Habu. Gekisashi became the first computer Shogi program to attend the human amateur Ryuo league, winning two games and ending up in the top 16 players in the tournament [6].

Game records have also been used to train evaluation functions in various games, including Shogi. The evaluation function of the Shogi program "Bonanza" is learned from game records [2]. The evaluation function of Bonanza uses the relative position of two pieces as feature. The number of its combinations is more than ten thousand. Bonanza uses game records to select the features to be used, and to adjust the weight of each feature. With this simple evaluation function and a fast search engine, Bonanza was able to win the 16th World Computer Shogi Championship. In Backgammon, the creation of an evaluation function which uses neural networks trained on game records was successful [10]. An automatic generation of an evaluation function in Othello has also been tried [5].

The method proposed in this paper uses game records quite differently from previous work, both in the approach and in the aim. Our methods statistically compare the different of game records of two groups. Either previous work treated game records as a single group, or the analysis was more focused on the meaning of moves played in the game, using the help of human interpretation. The goal of previous work using game records was purely to make the programs stronger, while our methods focuses on the difference between how the players from two player groups play the game. The difference could be anything, including the strength of the player and the playing style.

3 Discriminant Analysis

In this paper, we calculated a discriminant analysis [9] of significant features from game records. Three points should be considered when using statistical methods on game records: How to find the significant items? Is the result reliable? And what is its meaning?

In discriminant analysis, an explaining variable was defined as "1" for the discriminant target player, and "0" for the other players. The discriminant analysis was done twice, once for having to move, once for the opponent to move.

The Mahalanobis' *generalized distance* [9] was used as the discriminant function for the analysis. When there are two groups in a super space, the function divides two groups: by giving positive values for one group and negative values for the other. The function could be represented as a linear sum of weight w_i and the explaining variable x_i, as follows;

$$\sum_{i=0}^{n} w_i x_i (i = 0, 1, 2, ..., n) \tag{1}$$

The purpose of this discriminant analysis is to calculate w_i. The generalized distance is a distance between groups. Unlike Euclidean distance, the variance and the covariance are also considered. It could be used as the distance of two groups on a super space of variables. the generalized distance is the following D, where variable groups x, y follows the normal distribution, and the coefficient of the correlation is r.

$$D^2 = \frac{z_x^2 + z_y^2 - 2r z_x z_y}{1 - r^2} \tag{2}$$

z_x, z_y are calculated from the standard variation s_x, s_y and the mean values, $\bar{x}, \bar{y}$ of the variable quantities x, y from the following formula;

$$z_x = \frac{x - \bar{x}}{s_x} \qquad z_y = \frac{y - \bar{y}}{s_y} \tag{3}$$

The correlation r is expressed in the following formula. Each variable of x, y is represented by $x_i, y_i (i = 1, 2, 3, ..., n)$;

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}} \tag{4}$$

3.1 The Variables used in the Analysis

In this research, variables could be extracted directly from the game records. *How*, *where* and *when* were used. Simple items did not become variables, for example the progress value, and were multiplied with the other items. We guessed whether the item was important and how much. We used the features, because the arbitrary factors of the programmer were minimized. We found how the simple data of the game records was used to discriminate two groups.

The discriminant analysis was used with variable quantities to categorize from the features, for example the moves and the position. We found how the variables discriminate the two groups.

3.2 Selection of Variable by the Stepwise Method

No variables were equally effective for the discrimination. So, we selected the important variables by the stepwise method [9]. The variables were selected by removal or substitution of candidates. After the step, the variable vector was significantly changed or not. The variable P is based on the partial F value of the removal candidate variable *Pin* or the substitutive candidate variable *Pout*. We removed variables that either had no effect in discriminating the two groups or gave effect to both groups, with a significance level of 5%.

Although selected variable are effected by other independent variables, the variables selected has the significance proved to be at least 5%, considering the groups used to select the variables.

4 Comparison between Top-Level Human Players

To see if our method is able to find interesting features, we first conducted experiments comparing Shogi Grandmaster Habu and other professional Shogi players.

The game records we took were from the "Shogi Yearbook CD-ROM version" by Japan Shogi Association from 1985 to 2005. There were 9342 games in the records, 1013 in which Habu played.

In our previous work [11], we showed that Habu moved *Ryu* (promoted Rook) and *Tokin* (Promoted pawn) more frequently, and *Gyoku* (King) and *Kyo* (Lance) less frequently than the other players. There were also differences seen in positions where Habu moved his pieces, and the length of the game.

By using the method proposed in this paper, we were able to show not only what the *difference* is, but also how different each *difference* is.

We made two experiments for each feature. The first is to perform a discriminant analysis to estimate if the first player is Habu, and the second is to perform a discriminant analysis to estimate if the second player is Habu.

4.1 The Type of Pieces Used

We used the frequency of the type of pieces used in the games, and performed the discriminant analysis. We used 28 variables, since there are 14 types of pieces, and each of them was distinguished if being played by the first player or the second player. Although there was a possibility of a declining threshold, because the games of Habu were short overall. This is not a problem as this would appear as features. The result is in Table 1.

Table 1. The result of discriminate by the number of moves

	Generalized Distance	Discriminant Rate	Variable
First Player	0.44472	56.67%	28
Second Player	0.46787	58.31%	28
First Player Selected Variables	0.36473	54.96%	3
Second Player Selected Variables	0.41884	56.99%	6

The player to play second had a high recognition rate and a wider distance between the groups. These values were used in combinations with the latter experiments.

4.2 The Positions Played

In the next analysis, positions played were used as the feature. There are 81 squares from which pieces can be played. As moves can be played by the first or the second player, there were 162 variables. The result is in Table 2. Compared to the previous experiment, there are more variables.

Table 2. The result of discriminate by the number of position

	Generalized Distance	Discriminant Rate	Variable
First Player	0.84649	65.76%	162
Second Player	0.85035	66.07%	162
First Player Selected Variables	0.56405	60.51%	18
Second Player Selected Variables	0.53107	58.27%	12

When the variables were not selected, the discriminant rates were more than the rates of the number of used pieces. The generalized distance between groups increased because there were mores dimensions in the super space. As for the player to play first and the player to play second, the generalized distance and the discriminant rates were improved to the variables.

Now we compare the results of the analysis performed with the selected variables in this experiment. The result of using all the variables was in the previous experiment. These experiments were able to achieve the higher discriminant rate with fewer variables. This result shows that the positions played are better than the pieces played as the features distinguishing the two players.

4.3 The Number of Piece Capturing Moves and Check Moves

In this analysis, the frequency of a check move and the frequency of a piece-capturing move were used as features. The discriminant rate for a piece-capturing move was 54.20%, and the generalized distance was 0.31201 for the player to play first. The discriminant rate was 51.93%, and the generalized distance was 0.21894 for the player to play second. Although we conducted variable selection, as there were only two variables, no selection was made.

Next we computed the results which discriminated each group by the number of check moves. The discriminant rate was 47.18%, and the generalized distance was 0.18723 for the player to play first. The discriminant rate was 45.46%, and the generalized distance was 0.11839 for the player to play second. The number of check moves for the player to play second was not transferred to the discriminant analysis. The results of the discriminant analysis with the number of check moves were not good enough to discriminate the two groups with respect to the other items.

5 Difference between Human and Computers

In the following experiment, we compared the difference between human players and Shogi-playing programs. The game records of human players were taken from games played online by (mostly) amateur players of various skill levels, in "Collection of 240,000 games from Shogi-club". The game records of Shogi-playing programs were generated by games between commercial Shogi software.

5.1 The Length of the Game

We paid attention to the length of the games between the humans and the computers. Most programs play useless check moves when they are losing the game, as novice human players might make a mistake in defending the check move. Also, most programs do not declare draws. This tends to make computers to play longer games than human.

This is an important difference between human and computers, making games played by computers look unnatural.

5.2 The Type of Pieces Used

We used the frequency of the type of pieces used in the games, and performed the discriminant analysis. The results are shown in Table 3.

Table 3. Discriminant Analysis Using Pieces

	Generalized Distance	Discriminant Rate	Variable
Before Variable Selection	2.24850	87.13%	28
After Variable Selection	2.24575	87.19%	22

The generalized distance was 2.24575 and the discriminant rate was 87.19%. It is interesting to see that the generalized distance and the discriminant rate did not largely change by using more variables. This showed that the 22 selected variables were important in discriminating humans and computers. The moves extracted as features include King moves and moving promoted pieces.

5.3 The Positions Played

In the next analysis, positions played were used as the feature. The results are shown in Table 4.

The generalized distance was 1.36321. This is smaller than the previous experiment using piece types, showing that there is a small difference between computers and humans in the positions played.

Table 4. Discriminant Analysis of Position Frequency Used

	Generalized Distance	Discriminant Rate	Variable
Before Variable Selection	1.36321	77.03%	162
After Variable Selection	1.30474	76.37%	53

The results before and after variable selection were not so different. For the results after variable selection, 53 variables were selected. While this is more than the previous experiment using piece types, the discriminant rate is less. This result is the opposite of the results in the experiment comparing human players.

5.4 The Number of Piece Capturing Moves and Check Moves

In this analysis, the frequency of check moves and the frequency of piece-capturing moves were used as the features. The result are shown in Tables 5 and 6.

Table 5. Discriminant Analysis of Capturing Moves

	Generalized Distance	Discriminant Rate	Variable
Before Variable Selection	2.50986	92.06%	2
After Variable Selection	2.50986	92.06%	2

Table 6. Distinction of Check Moves

	Generalized Distance	Discriminant Rate	Variable
Before Variable Selection	2.23229	86.45%	2
After Variable Selection	2.23229	86.45%	2

For both capturing moves and check moves, a high distance and discriminant rate was achieved, only with two variables. This shows that these two features represent very well the difference in playing style of computers and humans.

6 Discussion

In this section we discuss two major observations from our experiments. In Section 6.1 we discuss the influence of the number of variables used. Next, in Section 6.2, we discuss the key features discriminating between humans and computers.

6.1 Reducing the Number of Variables

Reducing the number of variables did not reduce the discriminant rate of the analysis largely. In the experiments, we have selected variables by the stepwise method, and performed the discriminant analysis with both all the variables and the selected variables. The ratio of the variable reduction and the change in the discriminant rate varied from experiment to experiment, but in the whole, we were able to reduce the variables without sacrificing the discriminant rate. This means that those variables represent the key features that show the difference between the two groups of players.

6.2 Key Features in Discriminating Computers and Humans

The key features in discriminating computers and human were quiet different from the features that showed the difference between Habu and the other players. The number of features was much less, including the length of the game, the frequency of capture moves, and the frequency of check moves. This shows that there is still a large difference in how computers play and how humans play Shogi. Also, some features directly represent well-known computer “quirks”, such as making useless check moves instead of resigning, when the computer knows it loses the game (which is, of course, not a good manner in human games).

7 Conclusion

In this paper, we have proposed a method to compare game records from the two different groups of players, in order to find the difference in how the players in each group play. The game records are compared using discriminant analysis, and as result, the features that discriminate most the two groups are selected as the key features that show the difference in the playing style of the two groups. We have made two experiments, one comparing the Shogi Grandmaster Habu and the other top-level professional Shogi players. The other compared human Shogi players and Shogi playing programs. Our analysis showed that the difference could be seen in when (according to the progress of the game) and how (the pieces type and the position) the game was played. This information can be used, for example, to select the features to be used in evaluation functions, both for making strong Shogi-playing programs and also to give natural playing styles to Shogi-playing programs.

References

1. Hair, Joseph F., Tatham, Ronald L., Anderson, Rolph E., Black, William. : Multivariate Data Analysis. Prentice Hall; 5th edition (1998)
2. Hoki, Kunihito. : Optimal control of minimax search results to learn positional evaluation. The 11th Game Programming Workshop(GPW2006) 78–83, (2006).(in Japanese)
3. Iida H., Uiterwijk J.W.H.M., Herik, H.J.v.d. and Herschberg.: I.S.: Potential Applications of Opponent-Model Search Part 1. The domain of applicability. ICCA Journal, Vol. 16, No. 4, 201–208, (1993).
4. Ito, Takeshi. : Thinking of Computers vs Professional Players. The 10th Game Programming Workshop in Japan 2005(GPW2005) 40–47, (2005). (in Japanese)
5. Kaneko, Tomoyuki., Yamaguchi, Kazunori., and Kawai, Satoru.: Pattern Selection Problem for Automatically Generationg Evalution Functions for General Game Player. The 7th Game Programming Workshop in Japan 2002 (GPW2002) pp.28-35, (2002).
6. Matsubara, Hitoshi. (ed.): The progress of computer Shogi 4. Kyoritsu published, (2003). (in Japanese)
7. Matsubara, Hitoshi. (ed.): The progress of computer Shogi 5. Kyoritsu published, (2005).(in Japanese)
8. Matsubara, Hitoshi. and Takeuchi, Ikuo. (eds.): Game Programming. Kyoritsu published, (1998). (in Japanese)
9. Ohtsuki, Tomofumi. : Extraction of “Forced Move” from N-gram Statistics. The 10th Game Programming Workshop in Japan 2005(GPW2005) 89–96, (2005). (in Japanese)
10. Tesauro, Gerald. : Temporal Difference Learning and TD-Gammon. Communications of the ACM, March 1995 / Vol. 38, No. 3.
11. Tosaka, Kosuke. and Matsubara, Hitoshi. : The feature extraction of players from game records in Shogi. IPSJ SIG Technical Reports(2006-GI-16) (2006) 9–16. (in Japanese)
12. Tosaka, Kosuke. and Matsubara, Hitoshi. : Feature evaluation factors of players from game records in Shogi. The 11th Game Programming Workshop in Japan 2006 (2006) 171–174. (in Japanese)
13. Tsuruoka, Yoshimasa., Yokoyama, Daisaku., and Chikayama, Takashi. : Game-tree Search Algorithm based on Realization Probability. ICGA Journal, Vol. 25, No. 3, 145–152, (2002).

Selfish Search in Shogi

Takeshi Ito[1]

[1]Department of Computer Science,
University of Electro-Communications, Tokyo, Japan
ito@cs.uec.ac.jp

Abstract. When playing intellectually demanding games, such as Shogi (Japanese Chess), human players do not perform exhaustive and monotonous search as computers do. Instead, a human player selects a move from a set of candidate moves based on intuition, and linear search. It turns out that a search process of an expert player does not necessarily coincide with the search process of a computer although both may arrive at the same move to be played. It shows that a human player is thinking within the limits of a search tree which he believes is the right tree. This implies that each human player may search a different game tree. I call such search process "selfish search" and discuss the main characteristics of this search process.

1 Introduction

If a human faces a complex problem with a large number of potential solutions and calculations, he will not always act logically. In gamble, public lottery, etc., we see a mechanism by which a bookmaker makes a profit probable. There the chances of winning are always overestimated by the human player. For example, in public lottery, some people are allured by ungrounded rumours (such as "it is the turn of the sales store to win"). Other people take superstitious actions grounded on fortune-telling. A fully different example of irrational human decision making in games can be found in the domain of the imperfect-information and the chance gambling game Mahjong. A recent study [1] evaluating game records of internet Mahjong concluded that some amateur players treat a professional's remark like a proverb and believe it blindly in spite of a lack of evidence to support the proposition.

Not only in games of chance, but also in combinatorial games like Chess, Shogi (Japanese Chess) or Go, a human decision is mostly not necessarily founded on rational thinking. Many players are tending to select a candidate move intuitively based on their knowledge and experience. The study of Chess by De Groot is a famous example [2]. In previous work, we conducted mental experiments in the domain of Shogi and replicated De

Groot's research on the human intuitive thinking process in the domain of Shogi [3,4,5,6].

Computer programs perform the decision-making tasks in combinatorial games such as Shogi in a quite different vein from human players. Here we have two opposing systems that deserve further research.

First, human experience cannot be treated as intuitive knowledge. So, we have to address the human search process in another way than by heuristics only. Second, the computer is arriving at a (best) move by the virtue that its search process is based on a rigidly structured search. In computer Shogi, the technique of finding the best move happens by the alpha-beta game-tree search. It is based on an evaluation function that is widely used. The approach is known as "a search-driven system" [7].

Computer Shogi avoided incorporating human heuristics as much as possible (since intuition is considered to be more important than experience), and has developed the technology for searching deeply, fast and wide [8]. In the last decade we see that Shogi programs are becoming stronger with the progress of the hardware technology. The playing strength of the top class Shogi programs is said to roughly equal the strength of 6-dan amateur players. It reaches the level of professional players [9].

However, strong Shogi programs play sometimes moves which are perceived as *unnatural* by human players. As mentioned above, a significant difference exists between the decision-making process of humans and computers. A reason is that computer programs, in spite of playing strongly, suffer from intuition. Their style of play may be uninteresting and not useful to study for a human player. This might be so because a player does not understand the meaning of the move played by the computer. Some programs are able to display a part of their search log to show the 'meaning' of a particular move. However, since the move-decision process is not presented completely, it may remain unintelligible as a whole, and the human player cannot understand it.

In order to show a decision-making process in a form understandable to a human, it is necessary to clarify the thought process that human players are performing. In this article, we show what kind of search process is carried out in the human thought process which generates a move when playing Shogi.

Based on interviews with human expert players and verbal protocol data of human players of various levels, we will discuss in particular the "mechanism of prediction" peculiar to human players.

2 Some Results from Psychological Experiments

So far, we performed psychological experiments on the cognitive processes of human beings who play Shogi. In the current experiment we investigate the human thought process. We showed the same next move to beginners as well as to top professional Shogi players. Then a think-aloud protocol was recorded. As the result, we found that the thought process of the human who plays Shogi is divided into the following five stages [4].

(1) The stage of understanding a position
The stage of seeing the arrangement of the pieces on the board, and understanding what kind of position it is in.

(2) The stage in which candidate moves are generated
The stage which generates candidates for the next move based on intuitive knowledge.

(3) The predicting stage
The stage which predicts the move to be played based on candidate moves.

(4) The stage of comparing and evaluating moves
The stage that synthesizes the evaluation by intuitive judgement and by prediction; it compares the candidate moves.

(5) The stage of determining a move
The stage of determining a next move as a result of comparison.

The five stages of the human thought process were observed at all player of every Shogi skill ranging from the 10-kyu of a low level player to eight or more-dan of a top professional player. Although there is a difference in the contents of the thinking process, it has turned out that the thought process goes through five stages. This framework of thinking that is peculiar to a human being is called "Player Script".

If the contents of the thought processes are investigated in detail, it will turn out that the following three characteristics of human thought processes are seen in the generation stage of the candidate moves and the stage of prediction.

(1) Reducing the set of candidate moves to a few moves

Figure 1 expresses the average number of moves mentioned as candidate moves at the time of determining the next move and making the human being to think freely on the next move [6]. In the figure we see that the amateur 1-dan (middle-level players) raised many candidate moves. In average they still arrived at five whereas there are only three candidate moves. Since the problem given here is the position where a legal move exceeds 100 hands from tens hands, it turns out that a part of mere lawful hand is examined.

Considering the problem that a hand with pieces easily exceeds the number of 100 moves, it turns out that players are bound to examine only a part of the legal moves.

It is assumed that human players cannot consider many matters in parallel. Moreover, a human does not count all legal moves and so it differs in this respect from a computer. In practice it turned out that the human search process is particularly used for candidate moves when only a few moves exist. The move chosen is then evaluated by intuition.

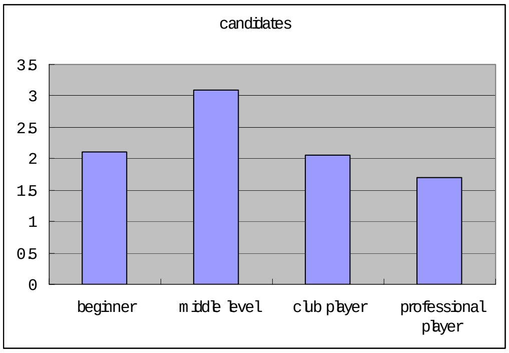


Fig. 1. Average number of candidate moves on next-move test.

(2) Predicting linearly

Given is the problem as shown in Figure 2. By the time he had to choose a next move the Shogi player under consideration was made to utter his decision-making process; the contents were recorded and analyzed [6].

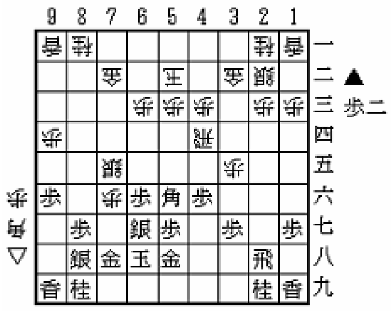


Fig. 2. A problem of next-move test experiments.

Figures 3-6 express by search trees the move that players of a certain Shogi strength predicted as the next move in the position of Figure 2. Although there is a great difference in the contents of thought, it turned out that the form of the search trees has almost the same form. Moreover, although there are some branches with candidate moves, the tree predicts almost linearly.

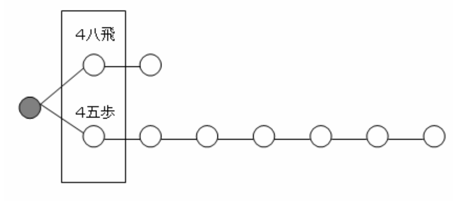


Fig. 3. A search tree of a novice.

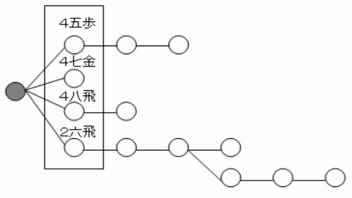


Fig. 4. A search tree of a mid-level player.

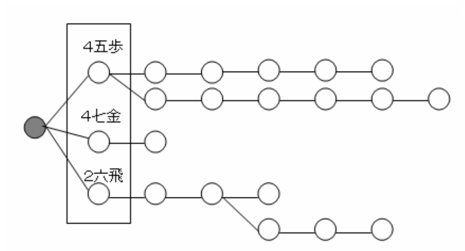


Fig. 5. A search tree of a club player.

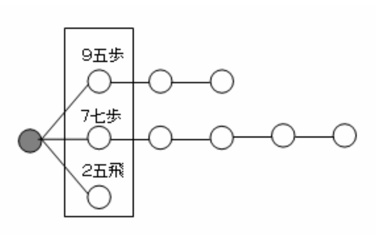


Fig. 6. A search tree of a professional player.

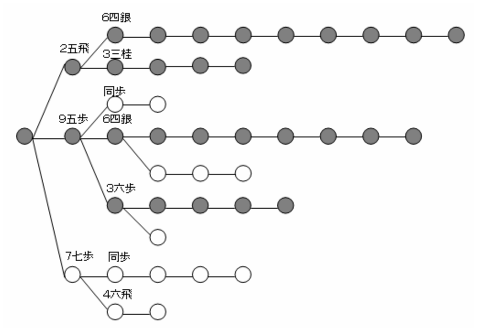
We believe that a move generation system for mimicking the thought processes of a human Shogi Player who advances and arrives at some point does so linearly. In computer Shogi, the candidate moves are all the legal moves, and a game tree is constituted from a prediction by assuming all legal moves by Black and White. The program searches by the minimax method using a static evaluation function which assesses the advantage or disadvantage of the position. However, it is clear that a human does not do minimax calculations like a computer. A human player predicts almost linearly. He starts at the intuitively proper move, and predicts to confirm whether the future position becomes really good for himself.

(3) A Professional Shogi player's prediction is different

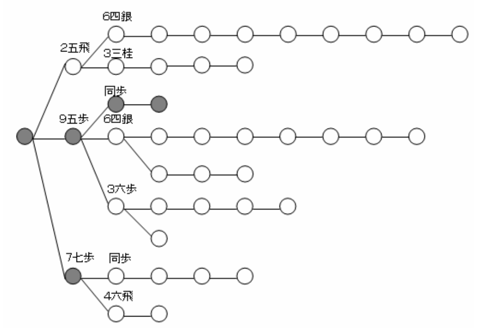
Figure 7 compares the thought process in the same search space when four top professional Shogi players are given the problem of Figure 2. ●-nodes are positions which the professional Shogi player actually mentioned, and ○-nodes expresses positions which he did not refer to. A result is that each professional Shogi player is predicting a node which is different from the predictions by others. In the figure, a difference is seen by following the search process. It happens also among top professional Shogi players.

Although it needs a special explanation, I attempt to deepen our understanding. I supplement therefore some details and a variety of thought processes. It seems that the professional player 1 searches deeply about "25-rook" and the candidate move of "95-pawn", and thought that "95-pawn" was sufficient at the beginning. Moreover, he has made reference deeply about "25-rook" which may become better. The professional player 2 assessed the candidate move "77-pawn" to be slow, and thought of "95-pawn" to be the main prediction. Unlike the professional player 2, the professional player 3 started with a reference to the bad formation called "wall-silver (88-silver)", and he opined that it cannot be good except when "77-pawn" is played. Though the moves which may exist to be a candidate, "25-rook" and "77-pawn", were considered, the professional player 4 thought that "95-pawn" was sufficient, and so he was predicting the move broadly.

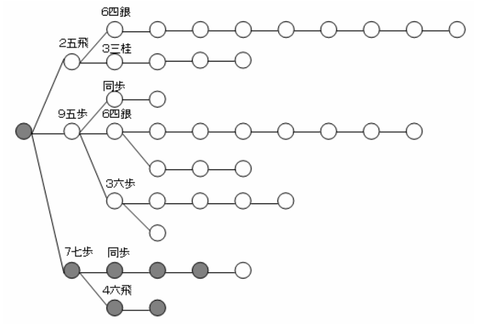
Thus, in some difficult positions, it became clear that predicting the candidate move was a different process for a top professional player and for other (average) players.



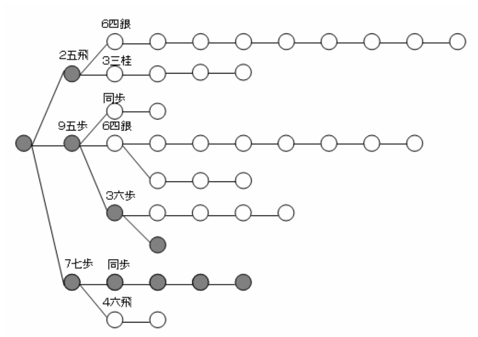
A thought process of professional player-1: 25-rook



A thought process of professional player-2: 95-pawn



A thought process of professional player-3: 77-pawn



A thought process of professional player-4: 95-pawn

Fig. 7. Search trees for professional players.

These results indicate that each player has predicted the space of each search tree in relation with each judgment criterion, and has chosen the move accordingly.

3 Selfish Search Based on Psychological Restrictions

As seen in Section 2, a candidate move is extracted and then the search process peculiar to a human being is made. The process predicts linearly and is called "Selfish Search". So it is different from the search by a computer. We believe that this "selfish prediction" is a thought process that is peculiar to a human. It is bounded by psychological restrictions by the human being. The range of the target problem is narrow. If a problem is counted in all its possibilities, it is also possible that all the cases are performing a comprehensive search like a computer. For example, if it is a game with comparatively narrow search space such as Tic-Tac-Toe (e.g., by summarizing the symmetrical position or eliminating the moves in which it loses simply), it will be possible to narrow the search space sharply. Analogously, a human will also count all the possibilities, and he will try to deduce a conclusion. However, the difficult games (the games of Go, Shogi, Chess, etc.) which have been played for a long time have a search space in which a human cannot search easily. In the large game tree of such a search space, a human gives up all searching, performs a "selfish search" within the limits which focus on some moves. He then can assume by "intuition", and is considered to determine a next move. Figure 8 expresses the number and speed of the prediction in relation to Shogi skill. The graph is from the result of an above-mentioned psychological experiment. According to this figure, it turns out that the speed of searching becomes quickly, so that the Shogi skill becomes high, and the number of searching is increasing according to it. However, the speed with which a top professional player searches is about ten moves per minute. The space which can be searched within a limited time is considered to be about at most some hundred moves. Thus, it can be said that there is a psychological limit in a human's search speed.

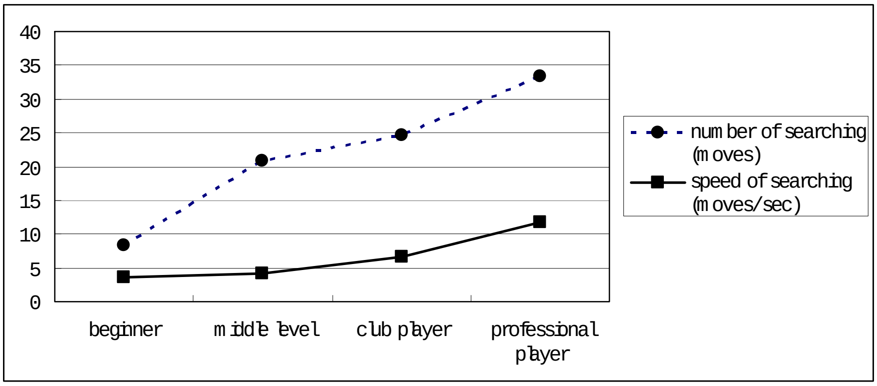


Fig. 8. Number and speed of searching as a function of Shogi strength.

Besides, a comparison of the search tree described by (2) in Section 2 shows that a human is performing as few linear predictions of a branch as possible. Furthermore, a human never calculates minimax-search for the all search trees with generating parallel branches from a certain node. The human advances by one reading at a time; many linear prediction results are compared simply, and a next move is determined so that it may become a straight line by a certain fixed view. In other words, a human can say that he is poor at performing a parallel prediction, and that he can perform only sequential predictions.

Such "a limit of searching speed" and "a limit of parallel search ability" are limits of functional calculation abilities of the human. I call this limit "psychological restrictions". If it thinks as mentioned above, it can be said that human thought is bound and prescribed by these psychological restrictions. "Selfish Search" which the human player is performing can be considered to be the optimized thought method in the "psychological restrictions" of the human.

4 Functions of Selfish Search

It has become clear for top professional players that they predict the positions differently, or even have obtained completely different search trees, from the results of (3) of Section 2.

It is considered that a human has a sense of evaluations (called a *Taikyokukan*) over Shogi based on his empirical knowledge. The candidate move was generated according to the *Taikyokukan*, linear search was performed, and a next move is determined. Thus, a player hurls a mutual *Taikyokukan* at each other and is advancing the game. If a *Taikyokukan* is different, the world which is in sight by it is also different. The result serves as victory or defeat. This experience serves as the cause to adjust one's *Taikyokukan* again, and serves as acquisition of a new *Taikyokukan*. The *Taikyokukan* which changed is used efficiently in the next game.

In the field of cognitive psychology, the process of study is caught with the process of a change of a concept. Here, it can also be put in another word as a "concept" being a

Taikyokukan which the player to Shogi has. Hatano and his co-workers explained that the concept (*Taikyokukan*) felt the necessity for that change, and served as motivation to update when a discord arises [10].

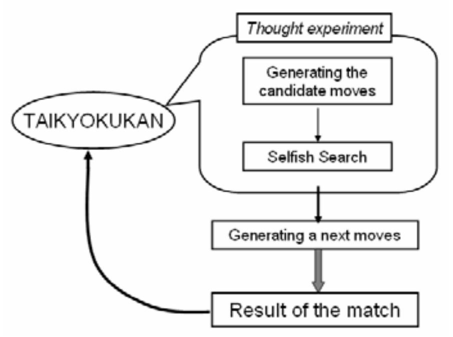


Fig. 9. The process of *Taikyokukan* change.

Figure 9 expresses the process in which a *Taikyokukan* changes, by a diagram. A candidate move is generated by the *Taikyokukan* and selfish search is performed in the head (as a thought experiment). A move is chosen based on selfish search, it is repeated, and it results in the outcome of the match. Then, an actual result is made to associate with the selfish search which the player assumed. When a difference is accepted there, the player feels discord and the change of *Taikyokukan* is pressed for him.

Thus, it is considered to be an important factor in which a difference with the actual game is given sensitive that a thought experiment is conducted in the style of selfish search. As the result of (3) of Section 2 shows, it often happened that the player and the opponent's selfish search were not corresponding with each other.

A player is performing "selfish prediction" in the psychological restrictions described in Section 3, if he feels sharply the difference in thinking between him and the others, and can say that he has reached a state in which it is easy to change a broad self-perspective. A player is performing "selfish search" in the psychological restrictions described (in Section 3), if he feels sensitive to the difference with thinking of him and the others, and has reached a state in which it is easy to change a self *Taikyokukan*.

If the above-mentioned is summarized, it can be said that there are the following two functions in "selfish search". One is a function called a thought experiment, by carrying out a simulation in the head; it is the function which tries to choose a better move. Another is a function to which the change of a self *Taikyokukan* is urged.

It can be said that selfish search is conducting the thought experiment which carries out the hypothesis testing of its sense of evaluations by considering a function of the former. In a certain position, as a result of advancing the position in one's sense of values, it is possible to assume previously what kind of position it becomes. It is considered to have appeared as a phenomenon of linear prediction. As a result of evaluating the assumed position, when it was judged that it is not desirable, the subject was actually considering another candidate move. Generally, the space of the search currently performed by selfish search is not comprehensive search like computer Shogi but only thinking of the very narrow space bound by the sense of values of the player. In amateur matches, there are also many predicting omissions and oversights. A game is more common as this prediction not to go on. However, the bad move which can be assumed by reading a few can be avoided, and the probability which chooses a bad move becomes lower by repeating this. Therefore, although many players know that it is the restrictive space, they predict. Besides, it is thought that the prediction capability also improves in proportion to Shogi skill so that the capability to read quickly may be needed and it may see as a result in Figure 8.

A function of the latter is the effect of study by selfish search. For example, when a certain result is shown, it is known for touching the result, after fully assuming beforehand about the result, and touching only a result, without carrying out such assumption that a mighty difference will arise in the learning effect about the matter. Although such a psychological phenomenon is called “study should care about”, the deep recognition to the position arises by repeating the process of the hypothesis testing by this prediction, and it is thought that it has mighty influence on a learning effect. Mr. Yoshiharu Habu (who is a famous Shogi top professional player) has described it as “the time he thought that it was most progressed became the professional Shogi player whose thinking time increased” in writings [11]. What more “learning set” came to be able to carry out will be considered to have led to the learning effect, when it has, time increases as one of the reasons of this and the time of prediction increases.

5 Proposal of Selfish Search System

In this section we first describe our proposed selfish-search system (Subsection 5.1), and then give an example of the working of our system (Subsection 5.2).

5.1 Composition of System

In advance of this selfish-search system, I have developed the knowledge-driven Shogi system (HIT: Human Intuitive Thought) imitating human intuitive thought [12]. This system can generate a move by describing the intuitive knowledge which the expert of Shogi has in the form of a production rule, without searching. In HIT, a score is first given to all the legal moves by the production rules prepared beforehand. About 400 production rules are given to HIT now. By applying these rules, a score is given to all the legal moves. Based on the given scores, the moves are sorted. The move which has the highest score is generated as a candidate move. A next move is determined by performing selfish prediction as shown in Figure 10 using the candidate move generated by HIT.

As shown in Figure 11, if a position is given, a score will be given to all the legal moves by HIT, and it will be sorted sequentially from high to low. It will become a candidate move if higher in rank than the candidate move or within a threshold α. If the candidate is only one, the move is the next move. If there are two or more candidate moves, they will be searched deeply. This search is repeated until the difference between candidate moves becomes below the threshold value β. This repetition realizes linear search. This linear search is performed to all candidate moves, an evaluation function estimates the position of each termination, and the move with highest score is chosen as the next move. Selfish search of humans is imitated by following a series of such procedures.

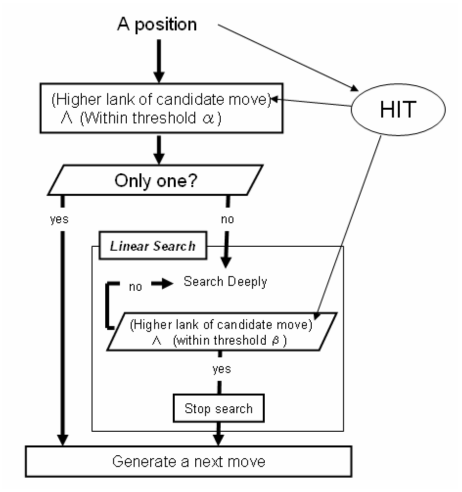


Fig. 10. Selfish-Search Algorithm.

5.2 An Example of Execution

The above-mentioned selfish-search system was implemented with HIT. The result in which the system solved the problem of Figure 2 is as follows. The execution condition was set to the threshold value α= 1000 and β= 400. A simulation was performed.

Figure 11 shows the above-mentioned conditions and is the result of performing the problem of Figure 3 by the selfish-search system. HIT receives the input positions and generates the higher-ranked five candidate moves: “45-pawn”, “45-bishop”, “48-rook”, “26-rook”, and “47-gold”. But, since the difference between the score of “45-pawn” and “47-gold” was larger than 1000 points, “47-gold” was removed as candidate. As for “48-rook” and “26-rook”, since the next move candidates were within 400 points, search was closed here. About “45-pawn” and “46-bishop”, the searching was continued until the next candidate moves were within 400 points. The evaluation function in the termination of four candidate moves was calculated, and “45-bishop” with the highest score was chosen.

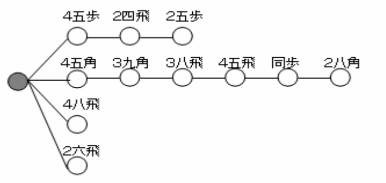


Fig. 11. An example of search by the selfish-search system.

6 Conclusion and Future Research

This article investigated the main characteristics of human search. The concept of "selfish search" was introduced to describe the decision-making process of a human player. We explained that "selfish search" is based on psychological restrictions. Moreover, we considered a function of "selfish search" in the middle stage of a game and formulated an adequate learning process.

Future research will address a knowledge-driven type of a computer-aided design using "selfish search" in combination with a learning-support system for human players.

References

1. Totsugeki-Tohoku: *Science of the Mah-Jong*, Kodansha (2004).
2. A. D. de Groot. *Thought and Choice in Chess*. Mouton Publishers, The Hague, The Netherlands (1965).
3. Takeshi Ito: A Cognitive Processes on Playing Shogi, *Game Programming Workshop in Japan '99*, pp. 177-184 (1999).
4. Takeshi Ito, Hitoshi Matsubara, and Reijer Grimbergen: Cognitive Science Approach to Shogi Playing Processes (1) – Some Results on Memory Experiments, *International Processing Society of Japan Journal*, Vol. 43, No. 10, pp. 2998-3011 (2002).
5. Takeshi Ito, Hitoshi Matsubara, and Reijer Grimbergen: Cognitive Science Approach to Shogi Playing Processes (2) – Some Results on Next Move Test Experiments, *International Processing Society of Japan Journal*, Vol. 45, No. 5, pp. 1481-1492 (2004).
6. Takeshi Ito: The Thought and Cognition on the Verbal Data of Shogi Experts, *IPSJ SIG-GI-12-2*, pp. 9-15 (2004).
7. Takeshi Ito: Thinking of Computers vs Professional Players – Contemporary and Future of Computer Shogi, *Game Programming Workshop in Japan 2005*, pp. 40-47 (2005).
8. Takenobu Takizawa: A Success of the Computer Shogi of the New Feeling by "Full Search" and Learning, and an Influence of the High-speed Algorithm, *IPSJ Magazine* Vol. 47, No. 8, pp. 875-881 (2006).
9. Takenobu Takizawa: Contemporary Computer Shogi (May, 2006), *IPSJ SIG-GI-16-1*, pp. 1-8 (2006).
10. G. Hatano and K. Inagaki: A theory of motivation for comprehension and its application to mathematics instruction. In T. A. Romberg & D. M. Stewart (Eds.), *The monitoring of school mathematics: Background papers. Vol. 2: Implications from psychology; outcomes of instruction.* Program Report 87-2, pp. 27-46. Madison: Wisconsin Center for Education Research (1987).
11. Yoshiharu Habu, Takeshi Ito, and Hitoshi Matsubara: Brains that Predict Future, *Shinchosha* (2006).

Context Killer Heuristic and its Application to Computer Shogi

Junichi Hashimoto, Tsuyoshi Hashimoto, and Hiroyuki Iida

Research Unit for Computers and Games
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan

Abstract. This paper proposes a new killer heuristic which we call the *context killer heuristic*. The original killer heuristic is based on a sibling relation between the positions considered or the blood-based similarity in the sense that all ancestors are the same. The context killer heuristic is based on the context-based similarity to pick up a killer move at any ply of a game tree. The "context" means a sequence of moves leading to a position just previously and is naturally extended as n-ply context. The context killer heuristic is implemented, where the mixture of 1-ply and 2-ply context killer moves are used for the dynamic move-ordering in a game-tree search based on the realization-probability search. Self-play experiments performed in the domain of computer Shogi show the effectiveness of the proposed idea. In particular, a program with the context killer heuristic played 300 games against one without it as Black and White under the same thinking time condition, and the score was 171-129.

1 Introduction

Since shogi is similar to chess, many techniques proven to be effective in computer chess have been adapted for shogi programs. Hence, strong shogi programs have a structure similar to chess programs. Basically, a shogi program builds minimax game trees explored by an iterative alpha-beta search. Shogi programs also make use of a variety of refinements, such as PVS-search, transposition tables, quiescence search and move-ordering [3]. Brute-force search has been very successful in chess. In recent years, the trend has reversed as improved algorithms and hardware have resulted in chess programs with highly selective searches. In shogi the need to be selective about which moves to search is much greater than in chess. It is a consequence of the larger move lists that have to be considered.

In this article we focus on the move-ordering technique called killer heuristic [2] in the framework of selective alpha-beta search, i.e., the realization-probability search [8]. We then propose a new idea called *context killer heuristic*, derived from a dynamic choice of killer moves based on a new concept of position similarity. The key point is *similarity* between two considered positions. The original killer heuristic focuses on the sibling positions to adapt a very good move found in other positions at the same ply, because such a sibling position is likely to be similar to the other positions considered. Since in the original killer move heuristic all ancestors including the parent node are the same in a game tree, we call it *blood-based* similarity. Our new idea focuses on a so-called *context-based* similarity, where we suppose that the same context may result in similar positions. With this notion of similarity, killer moves are to be dynamically picked up and adapted in a position which has the same context.

Section 2 describes the basic ideas of the context killer heuristic. In Section 3, experiments are performed in the domain of shogi and its results are discussed. Then concluding remarks are given in Section 4.

2 Basic Ideas of the Context Killer Heuristic

In this section we first sketch shortly the original killer heuristic and discuss the similarity between positions. Then we propose the so-called context killer heuristic, and some enhancements are described.

2.1 Killer Heuristic

The original killer heuristic [2] attempts to produce a cutoff by assuming that a move that produced a cutoff in another branch of the game tree at the same depth is likely to produce a cutoff in the present position. It means that a move that was a very good move from a different (but possibly similar) position might also be a good move in the present position. By trying the killer move before other moves, a game-playing program can often produce an early cutoff, saving itself the effort of considering or even generating all legal moves from a position. In this heuristic, killer moves come from the sibling nodes. It means that they have the same ancestors. Therefore, it is based on the so-called blood-based similarity.

2.2 Forward Similarity and Backward Similarity

Let us consider the notion of *similarity* since it is the key point to develop the so-called context killer heuristic. When we focus on judging similarity between positions, it can be of two types: *forward similarity* and *backward similarity*. In the forward similarity, a similarity relation can be determined in advance. On the other hand, such relation cannot be determined in advance in the backward similarity.

For example, Kawano proposed the definition of similarity between positions in the framework of mating search for shogi [4]. That similarity can be determined by adapting a mating sequence, found in a different position, to a sibling position considered. If it is successfully adapted, then it is confirmed that the two positions are in the relation of similarity.

Obviously, the original killer heuristic is in the class of the forward similarity. In some sense, Kawano's approach is a variation of the killer heuristic while adapting a sequence of moves instead of a move in the sibling positions. For search efficiency, the forward similarity is better than the backward similarity in finding a killer move to adapt it in other positions.

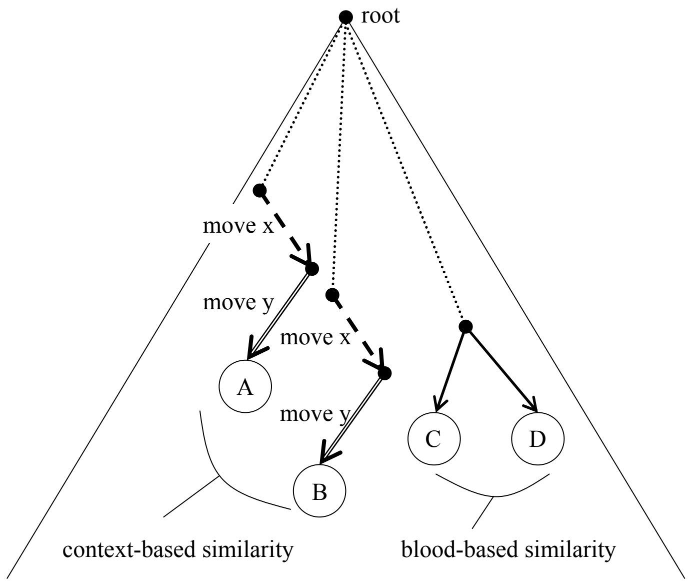


Fig. 1. Forward similarities.

Moreover, when we consider the forward similarity, we notice that there are some kinds such as *blood-based* similarity and *context-based* similarity. Let us show, in Fig. 1, the illustration of these forward similarities. If two nodes have the same parent in a tree, then they are in the relation of truly blood-based similarity. The ancestors are definitely the same and therefore they are brothers/sisters as shown in the figure as node C and D. Indeed, the original killer heuristic relies on the blood-based similarity to assume that these two positions must be similar.

Let us show, in Fig. 1, an example of context-based similarity where positions A and B are in this relation. Both of them have the same two-ply move sequence (say move x and y) just previously. The paths from the root node are different for the two positions. However, they sometimes are similar in the sense that the same strategy is being applied. The two-ply move sequence can be naturally extended to an n-ply move sequence. We observe that two positions become more similar as n becomes greater.

2.3 Context Killer Heuristic

We explain the context killer heuristic using Fig. 2. Suppose position C is a frontier node in a game-tree search, whereas position A has the same context of n-ply with position C, i.e., move p_1, move p_2 ... and move p_n. In the context killer heuristic, the best move found in position A would be stored as a killer move, which will be later adapted in position C that has the same context of n-ply with position A. When we pick up such a killer move based on the context-based similarity of n-ply to adapt in other positions, we call it n-ply context killer heuristic.

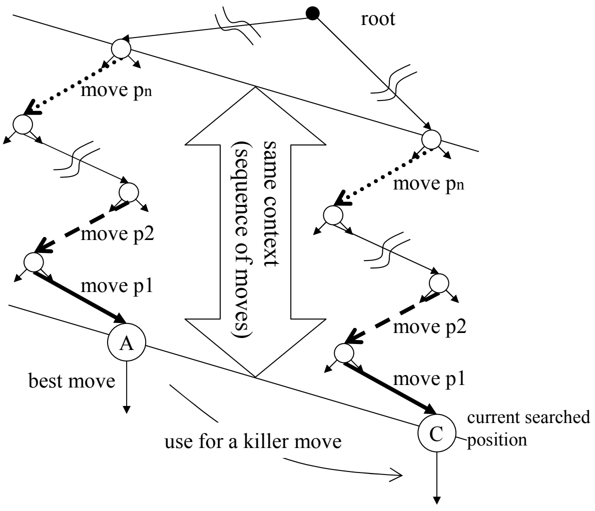


Fig. 2. Context killer heuristic.

2.4 Enhancement

When performing the context killer heuristic in practice, one critical issue is the expected high cost to identify the context-based similarity during the entire search. For this purpose a hash table is used, in which an entry is a best move found in a context of game-tree search. Let us show, in Fig. 3, a pseudo code of the context killer heuristic. The size of the hash table is ad-hoc. We use here a number larger than all possible legal moves in shogi (approximately 8000).

For the index (hashed context) of the hash table, we use the XOR of all hashed values of moves contained in the context. Since our priority is the speed for the procedure to identify similarity rather than the accuracy of the context identification, we are not seriously concerned with a conflict of the hash index. When any conflict occurs, the entry is simply overwritten.

```
/* hash table storing killer moves */
/* N is an ad hoc value (larger than the number of legal moves) */
MOVE context_killer_table[N];

/* calculate an index for the hash table from context(moves) */
int index(MOVE p1, p2, ..., pn) {
    return ( hash(p1) xor hash(p2) xor ...xor hash(pn) ) mod N;
}

/* store a set of a context and the best move on the context */
void set(MOVE best, MOVE p1, p2, ..., pn) {
    context_killer_table[index(p1, p2, ..., pn)] = best;
}

/* retrieve the best move for a context */
Move get(MOVE p1, p2, ..., pn) {
    context_killer_table[index(p1, p2, ..., pn)];
}
```

Fig. 3. C-like pseudo code of the context killer heuristic.

3 Experiments and Results

In this section experiments are performed to evaluate our proposed idea while using a test set of next-move problems and carrying out self-play games. First we describe the experimental design and then the results are discussed.

3.1 Experimental Design

For the experiments, the combination of the 1-ply and 2-ply context killer heuristic were implemented in our shogi program TACOS [5]. TACOS is one of the strongest programs and won the Computer Olympiad in 2005. It uses the alpha-beta algorithm in the framework of selective search, the so-called realization-probability search [8]. It also uses varieties of PVS, null-move pruning and internal iterative-deepening. Currently, move-ordering is made only in a static way. There are over 30 categories of moves, which are generated sequentially in the fixed order of the categories as follows:

1. hashed move from transposition tables,
2. moves capturing a piece which moved last,
3. the killer move (by blood-based similarity),
4. capture, check, escape from attack, defense, and so on.

The context killer heuristic is expected to increase the search efficiency, and then strengthen the performance level. We expect it with some significance since our shogi program uses iterative-deepening search. We adapted the context-based killer moves just after the blood-based killer move. If we obtain good results under this condition, it shows the two killer heuristics complement each other.

Since it is not easy to evaluate the effectiveness of the context killer heuristic itself due to the program structure, we perform two different tests: solving next-move problems and self-play games.

3.2 Results and Discussion

Let us show, in Table 1 and Table 2, the results of the performance tests. In these tables, 'context' means a version which uses the context killer heuristic.

Table 1. The results of the test using next-move problems.

	1st-dan	4th-dan
Context	67(105.2)	43(101.4)
Normal	63(102.0)	40(101.1)

Number of problems solved by each version and search speed(knodes/sec). Dan is a difficulty level of problem sets. Each set has 100 problems.

Table 2. The results of self-play games.

Context	Normal
171	129

Number of games won by each version.

The results of the test using the next-move problems show that the search speed goes up just a little while the number of problems solved goes up. The speed down by using the proposed idea, such as for memory accessing, can be ignored.

In the self-play tests, the program with the context killer heuristic won against the one without it by 171-129. The statistic shows that the program with the proposed idea became stronger.

From the self-play test performed, we are not sure that the proposed idea really can strengthen the program. However, from the next-problem test in addition to the self-play test we see that the proposed idea is really effective.

From the experiments performed in this paper, we know that the proposed idea affects the program's performance positively, but we do not know about how much it affects the search efficiency. We suspect that the proposed idea is useful for decreasing the horizon effect as well as increasing the search efficiency. This is because the horizon effect is a key factor for strengthening the performance level [3]. Moreover, the experimental results show that the similarity of positions is related to the context, i.e., move sequence, in the domain of shogi. Since the proposed idea does not use any game-dependent information, it should be possible to apply the context killer heuristic in other games.

4 Concluding Remarks

In this paper we discussed the notion of similarity based on the timing to indentify it, and proposed the concept of forward similarity and backward similarity. In this context the killer heuristic belongs to the class of the forward similarity, i.e., the similarity relation is identified in advance to adapt a killer move. One way is to pick up a killer move from the sibling positions like in the killer heuristic. Another one is to do so based on the context-based similarity, where a killer move is picked up dynamically with focus on the context of move sequences. This is what we proposed in this paper as the context killer heuristic.

The context killer heuristic has been implemented in a shogi program, where the mixture of 1-ply and 2-ply context killer move was incorporated. The performance experiments using a test set of next-move problems and self-play games confirmed the effectiveness of the proposed idea.

In this study the context killer heuristic has been tested in the framework of the alpha-beta algorithm based on the realization-probability search in the domain of shogi. One idea for future research is its application to other domains. Moreover, the proposed idea will be incorporated in a brute-force search program to see its effectiveness.

References

1. Akl, S.G. and Newborn, M.M. (1977), The principal continuation and the killer heuristic, Proceedings of the 1977 annual conference, pp. 466-473.
2. Huberman, B.J. (1968), A Program to Play Chess End Games, Technical Report no. CS-106, ph.D. thesis, Stanford University, Computer Science Department.
3. Iida, H., Sakuta, M., and Rollason, J. (2002), Computer Shogi, Artificial Intelligence, vol. 134, no. 1-2, pp. 121-144.
4. Kawano, Y. (1996), Using Similar Positions to Search Game Tree, in R.J. Nowakowski, editor, Game of No Chance, Cambridge University Press, vol. 29, pp. 193-202.
5. Nagashima, J., Hashimoto, T., and Iida, H. (2006), Self-Playing-based Opening Book Tuning, New Mathematics and Natural Computation, vol. 2, no. 2, pp. 183-194.
6. Schaeffer, J. (1989), The History Heuristic and Alpha-Beta Search Enhancements in Practice, IEEE Transaction Pattern Analysis and Machine Intelligence, vol.11, no. 11, pp. 1203-1212.

7. Tsuruoka, Y. (2005), Present and Future of Shogi Programs, IPSJ Magazine, Information Processing Society of Japan, vol. 46, no. 7, pp. 817-822. (in Japanese)
8. Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002), Game-tree Search Algorithm based on Realization Probability, ICGA Journal, vol. 25, no. 3, pp. 145-152.
9. Uiterwijk, J.W.H.M (1992), The Countermove Heuristic, ICCA Journal, vol. 15, no. 1, pp. 8-18.

Go

Checking Life-and-Death Problems in Go I: The Program ScanLD

Thomas Wolf and Lei Shen

Brock University, St. Catharines, Ontario, Canada,
twolf@brocku.ca,

Abstract. In this paper we introduce the program ScanLD (built on GoTools) which checks solutions of life-and-death problems for correctness. This is a task for which computer programs are especially useful. Not only can they do the computations, they can also do the handling of data in checking all moves of all solutions for optimality and reporting any errors that occur. Their refutation and their correction would be tedious and error prone if done with a computer, but interactively.
After discussing the different types of checks that are performed and giving some statistics resulting from checking a 500-problem tsume go book, some examples are given. A long list of mistakes that have been found is given in an on-line addendum to the paper.

1 Introduction

The computer Go program MoGo (see [2]), written by Sylvain Gelly and Yizao Wang, is built on the UCT algorithm invented by Levente Kocsis and Csaba Szepesvári ([3]). It has much recent success, for example, by winning the latest (March 2007) computer Go tournament on the KGS server (see [7]) and leading the continuously ongoing tournament on the 9x9 Computer Go Server (see [1]).[1] One might wonder whether there is still any purpose in using specialized Go software, for example, in analyzing life-and-death problems.

Although programs which use probabilistic methods (e.g., by evaluating positions using Monte-Carlo and searching the tree of moves using the UCT algorithm) are relatively strong Go-playing programs, they are not designed to *prove* statements.

In contrast, life-and-death programs like GoTools, give a result which is supposed to be exact (if the program is bug free) and the result consists not only of a best first move but also of a status of the position, including (if the status is ko) the number of ko-threats that one side needs to ignore in order to win. As a small study in [10] shows, the last few % of correctness cost an exponentially growing amount of computation time – for GoTools as a normal tree search algorithm – and we would claim even more so for probabilistic methods.

[1] The success of MoGo is the more remarkable both in that the size of its code is only a fraction of the amount of code that goes into established older programs and in that it scales well, i.e., it still has reserves and will probably be able to make good use of multi-core CPUs as they become more and more widely available.

A contribution which life-and-death programs can make is to pre-compute the status of enclosed areas and to store them in a data base which might be used by full game-playing programs. An example is a database of eyes surrounded by only one chain (monolithic eyes) which was computed recently (see [11]). As a by-product, a classification of strange positions was obtained which might be entertaining for Go players to look at (see also [11]). Other applications of GoTools include a program that generates new life-and-death problems (see [9], [6]) and an Internet application that solves problems online (see [8]).

In this paper GoTools is the workhorse behind the program ScanLD which is used to check thoroughly life-and-death problems including their solutions. This is described in the next section. It is followed by a description of how ScanLD was applied to 500 problems found in a Chinese tsume go book ([4]) which from now on we simply call 'the book'. In section 3 we show how many of these problems were solved by the program and how many gave problems. What might be of most interest to Go-players is section 4. It shows examples of problems where ScanLD uncovered non-optimal play in their solutions or other mistakes in the book. A long list of findings is given in [5]. Go-players can use the examples in this paper and the extensive list in [5] twice, once for solving the given problem and one more time by trying to find out what is wrong with the solution given in the book.

The life-and-death problems discussed in this paper are generally very interesting because at least some aspect of their solution was obviously missed by the authors, one of them being a professional 3-Dan player at the time of writing the book.

Although all move sequences computed from ScanLD in this paper and in [5] should be bug free, they may have 'features' resulting from performing a fast but simplified search in which, for example, seki is considered to be equivalent to life. This and other 'shortcuts' should be kept in mind when inspecting these sequences. For that purpose in the appendix all limitations of the programs GoTools (and thus of ScanLD) are listed for completeness.

2 About ScanLD

Life-and-death problems can be wrong in more ways than the obvious one that the shown winning first move is wrong. The following is a list of tests that are performed by ScanLD. The first 2 tests concern the problem itself, not the solution.

1. After ScanLD solves the problem it tests whether the side moving first can win, or at least reach a ko. If not then this is a less interesting problem.
2. Secondly, it determines the status for all possible first moves and thus finds the set B of all moves that give the same optimal result. If B includes more than one move then all moves in B and their likely follow-up sequences are shown. In that case the problem has no unique solution and is thus not a nice problem. We list them in section 4.3 if in the book no comment was made about the existence of another optimal solution.

3. The next test checks whether the first move of what is provided as the best solution, is among the moves in B. If that is not the case then all moves better than the best move given in the book are shown. Next, the status of the first move of each solution sequence is tested.
4. The program compares the given status of the first move with what the program had found earlier as the status for this move. If the given status of the solution is an unspecified ko[2] and if the program found ko as status too but a specific ko then this specific ko is from now on taken to be the correct status of the given solution, and is used as a reference when checking the next moves in the sequence.
5. Now the test of the solution sequences and the status of what the first move can reach begins. The following checks are performed for each given sequence and within each sequence for each move.
   The first check tests whether the move in the sequence is legal. The second check tests whether the move is optimal in the current situation, no matter whether the current status is the same as the original or whether it has changed due to earlier non-optimal play of any side.
   If a move is not optimal then the better move and the optimal status that it achieves are given. In addition, the program explains why the shown move is not optimal by showing how it can be countered with a likely follow-up sequence.
6. Sometimes a mistake by one side playing a non-optimal move in the solution sequence is compensated by a mistake from the other side afterwards or is followed by more mistakes by the same side. Therefore each solution sequence is checked to the very end to find all mistakes.
7. Finally, the order in which the alternative solution sequences are given in the book is checked and a comment is made if the side moving first achieves more in a later solution than in an earlier solution. This of course is not an error message. It only helps to sort solutions according to what they achieve.

In published problem collections variations of non-optimal moves are shown and discussed. These are of course not mistakes in the publication. Most often it is the wrong first move which is discussed and this would not lead to an error report by ScanLD. But if the discussed error consists of a later move then ScanLD would report this as a mistake. The current format of life-and-death problems as it is used by the programs GoTools and ScanLD does not allow us to mark single moves in a given sequence as known to be wrong. Therefore error reports of ScanLD have to be checked manually to see whether they coincide with the discussion in the publication.

3 Statistics

In this section some statistics for the success rate of ScanLD in finding mistakes in [4] is shown. The outcome of a test falls in general into one of the following categories which are listed again in table 1.

[2] This is mostly the case as very rarely is the precise ko-status given in problem collections, i.e., how many external ko-threats one side has to ignore in order to win.

a Because ScanLD can solve effectively only problems with a closed boundary, there are problems which can not be closed easily without changing their status (1.8%).[3]

– For another group of problems ScanLD closes the problems too widely or too narrowly. If the closure is too narrow the solutions are changed, if it is too wide then the problem is too complicated for the program. In these cases the boundary had to be adjusted by hand. The current routine used for closing a problem has still some room for improvement. An idea to be implemented in future is to use the solutions (if they are given) for constructing the enclosure such that the additional stones do not interfere with any move in any solution. On the other hand to make enclosures perfect is ultimately as hard as solving the problems. When ScanLD reports a mistake, it also prints the original position and the position after the closure to allow a quick inspection and if necessary a re-run with a human-improved enclosure if it was too narrow or too wide.

b Although most problems are fully checked in 10-30 min, others (33.4%) take much longer or are simply too hard to be solved and checked by ScanLD in a reasonable time of currently 1 day. The 500 problems have been divided into 25 groups and each group was run in batch mode on a separate node of a Pentium 4 Beowulf cluster.

c For another group (48.6%) of problems the program confirms all solutions in the book in the sense that all moves in all solutions belong to the set of optimal moves in each of the situations.

d The remaining problems where the program identifies a mistake in the book, could be categorized according to whether the mistake changed the status from life to death, or ko to life, etc. but we have them in one category when the status changes to life or death (4.2%) and ...

e a smaller group (2%) of problems where a ko is overlooked which prevents complete defeat.

f In a relatively large group (8.4%) of problems the non-optimality of moves is only minor in that the number of necessary external ko-threats changes that are needed to win a ko or the side changes which needs ko-threats. Although being less dramatic, such errors can make a difference between winning or losing a game too. Due to their relatively high number these problems are not included in this paper and not in [5].

g Finally we list a group (1.6%) of problems in which there is nothing wrong with the provided solutions except that the problem has more than one first move giving the same optimal result and this has not been mentioned in the book.

– An extra service for which no statistics has been made in this paper concerns problems for which solutions are missed that are not optimal but that are better than the worst solutions discussed in the publication. For example, if

[3] Such problems would need stones added outside the weak boundary to build life outside which is worth to connect to and the whole situation would have to be enclosed by one big boundary which the program can not do automatically.

● has one move that kills and moves that lead to life discussed in the paper but not existing moves that at least give a ko then this leads to a comment by the program.

Table 1. Statistics of the 500 problems in the book

Category	**Problems**	**Percentage**
a. boundary not easily closed	9	1.8%
b. timed out	167	33.4%
c. agreement	245	49%
d. changes status to life/death	20	4%
e. missing ko	10	2%
f. change of # of ko-threats	42	8.4%
g. no bug, more than 1 best move	7	1.4%

Checking the 500 problems from the book put the program to a rigorous test because for each problem the book gives 3-6 solutions and for each solution about 5-20 moves, each leading to a position that had to be solved in order to check whether the move in the sequence is optimal. In this process 3 bugs were found in GoTools that were corrected. More changes were made to the interface of ScanLD, i.e., how it reports discrepancies and moves which counter those on the book.

The numbers in the table reflect the situation as of April 2007. We intend to have a closer look at the relatively large fraction of unchecked ('timed out') problems. We will try give them more time, use more memory to allow a larger hash table and hand optimize the closure of the boundary to reduce their percentage. Findings will be used to update the online available paper [5] of all mistakes that have been found. Other outstanding work includes an improvement of the procedures that enclose an open problem although this is not too urgent as a closure by hand is not a problem and is done quickly.

4 Problems with Problems

As outlined in the previous section, there are three categories of mistakes which will be discussed in the following three subsections. In all cases the first diagram (like diagram 1 below) shows the original problem as given in the book without extra boundary closure as added by ScanLD.

The second diagram (like 2) shows a solution with non-optimal play from the book. If the book assumes that all moves in the sequence are correct (in this book solution 1 for each problem) then the task is to find any error as stated, for example, underneath diagram 2 is written 'Find any error'. If the second diagram discusses a move, say ❸, which is known in the book to be wrong then we discuss this problem only if there is at least one more mistake following ❸ that is not mentioned in the book. In such a case the caption underneath the second diagram would say 'Find error after ❸' with the understanding that it could be move ④ that is wrong or any other move, like ❼. Such mistakes are documented in the follow-up article [5]. Here we have space for only one example in each category.

The third and possibly further diagrams each correct one move from the solution sequence in the second diagram as in diagram 3.

To make the paper self-consistent and useful for the reader, for each problem a solution with correct (optimal) play should be given. If the corrected diagrams do not already include one with optimal play (because one move in the sequence is known in the book to be wrong and then the corrected diagrams will contain this move too) then another (last) diagram with optimal play is added. This extra last diagram can be solution 1 from the book (if it was correct there) or a correct solution which was missed in the book, as in diagram 4. The extra last diagram can also show an additional solution if the problem has more than one best move, as diagram 12.

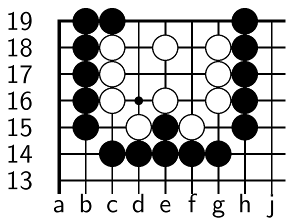
Go players could use this section for practice twofold: first, by covering all diagrams except the first one and trying to solve the problem on the left and a second time by uncovering the second diagram and trying to find the mistake in the solution sequence.

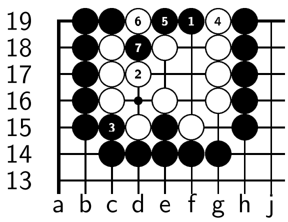
Most often a single diagram can only illustrate but not prove the statements that are made. In order to verify interactively comments in this paper the reader can solve and explore problems online at [8] but has to first close the boundary suitably.

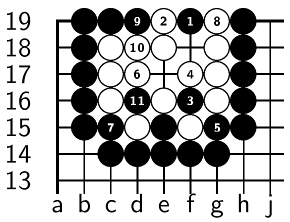
4.1 Mistakes changing the Status to Life or Death

For all problems in the book ● plays first. What we call 'solution 1' below is the first solution in the book which always is supposed to be the solution where both sides play optimally. Later solutions typically discuss non-optimal play.

Problem 33, solution 1


● to move.
Diagram 1.


Find any error.
Diagram 2.


⑫ @ ❸
Diagram 3.

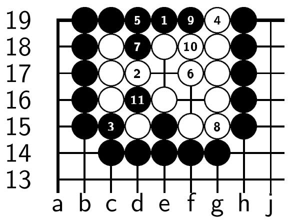


Diagram 4.

In Diagram 2, ○ needs to win a ko at d19 to live. But ○ can live by playing ② on e19 as shown in Diagram 3. The only ❶ that reaches at least a ko is shown in Diagram 4 and is missed in the book too.

4.2 Mistakes changing the Status to Ko

In the problems of this subsection moves are missed which would provide a ko for the side that otherwise would lose. We give one example; more are listed in [5].

Problem 181, solution 1

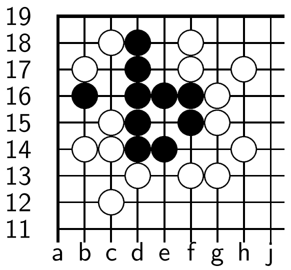
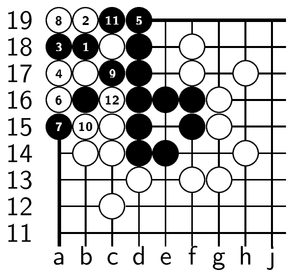

● to move.

Diagram 5.


⓭ @ ❶

Find any error.
Diagram 6.

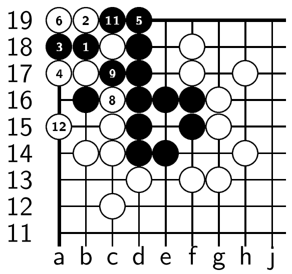

❼ @ ❸ ⑩ @ ❶

Diagram 7.

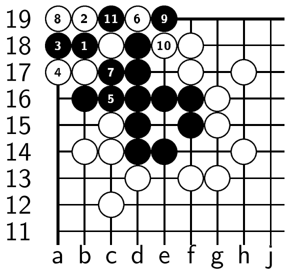


Diagram 8.

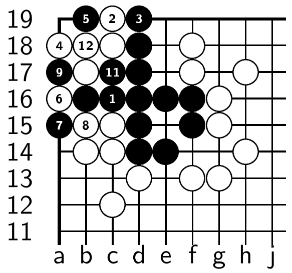

⑩ @ ⑥ ⓭ @ ❾

Diagram 9.

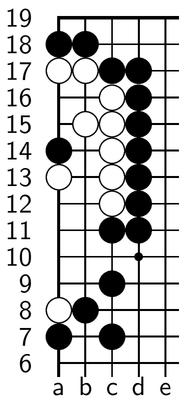
In Diagram 6 the move ⑥ is not optimal as it enables ● to live. Better is ⑥ on a19 as shown in Diagram 7 which kills. One move earlier, ❺ is an error too as it could have prevented death by playing on c16 in Diagram 8 leading to a favourable ko for ●.

A slightly better and optimal play of ● is shown in Diagram 9 where ○ needs one more ko-threat to kill. The solution in the last Diagram is equivalent to solution 3 in the book which due to the above mistakes is described there falsely as non-optimal.

4.3 Problems with more than one best Move

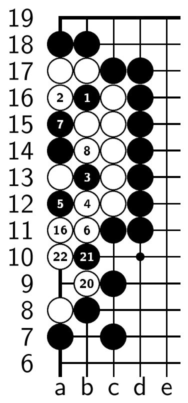
Some problems in the book are somewhat less suitable for a tsume go collection because they have at least two equally good first moves. If there are any mistakes in their solutions in the book then these problems have been discussed in the previous sections. Here we give an example of an error-free problem with non-unique solution. The extra winning move is often a strong threat which has to be answered before the best move from the book is made. Another source of multiple best moves is the situation where the order of ❶ and ❸ can be reversed.

Problem 81



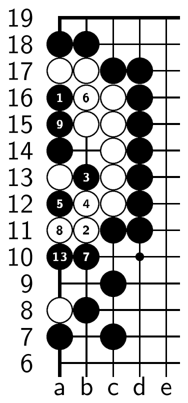
● to move.

Diagram 10.



❾ @ ❶ ⑩ @ a13 ⓫ @ ❸
⑫ @ b17 ⓭ @ a17 ⑭ @ a13
⓯ @ ❸ ⓱ @ a13 ⑱ @ ②
⓳ @ a13 ㉓ @ a14

Diagram 11.



⑩ @ a13 ⓫ @ ❺
⑫ pass

Diagram 12.

In addition to ❶ on b16 in Diagram 11, ❶ on a16 as in Diagram 12 also reaches the status that ● needs two external ko-threats to win the ko although in Diagram 12 ○ can afford to pass one more time (which the computer program ScanLD does not take into account). The moves 1-10 in Diagram 11 are from the book, the extra moves are from the program to show that ● needs in total two external ko-threats to win.

5 Summary

The emphasis of this paper was not to judge the quality of a specific publication on life-and-death problems but to prove the usefulness of the programs ScanLD and GoTools. Like every text to be published nowadays would always be run through a spell checker first, equally life-and-death problems should be checked by a computer program before being published. Also for readers that are not so strong it is advantageous to have a computer program which provides solution sequences to the very end and which answers any refutation attempt. This is often necessary to realize all the purposes a particular move has, they do not become obvious in a single variation.

The book [4] is the first one we checked. It includes a huge amount of information. The 500 problem diagrams should have not more than one best solution. It also includes 1990 solution diagrams with on average 8 moves, giving 15920 moves, of which about (1990 - 500) are identified in the book as wrong, leaving 14430 moves that could be wrong. If we ignore the problems with multiple best first moves but count multiple errors in one solution sequence then we found 83 wrong moves. As we checked about 65% of the book this gives an error rate of $83/(0.65 \times 14430) = 0.88\%$ for each move which seems very accurate to us.

A Disclaimer

The following is a list of either deliberate restrictions or of non-intentional weaknesses that GoTools, and thus the program ScanLD still have.

- By performing a search with only two possible outcomes (win or loss), GoTools finds a group to be alive if it can not be killed, i.e., it does not make a distinction between unconditional life and seki or a double ko life.
- When comparing the values of different kos and thus the quality of different moves leading to different kos, then two measures are important which characterize the cost for that player, say player A, to win the ko, who would otherwise lose the fight if no external ko-threat were played. One measure is the number of external ko-threats that A needs to have more than B. This number is determined by GoTools. The other measure is the number of passes/tenuki (plays elsewhere) that either side can afford in addition to the fact that A has to play the last move if A wants to win. This measure is not determined by GoTools currently. Ignoring this number would be correct if playing elsewhere had no value, such as at the very end of the game.
- When evaluating a ko position, the number of external ko-threats that is available to the loser is incremented successively until either some limit is reached, or until the loser wins. In the computer runs for this paper this limit is set to 5 because a position in which one side needs to ignore 5 ko-threats may as well be counted as an unconditional loss for that side.
- Open problems have to be enclosed before GoTools can solve them. If the enclosure is too wide then the problem may be too hard to solve. If the enclosure is too narrow then it may affect the status of the different first moves. For example, it happens currently that ScanLD finds additional best moves that kill but a closer human inspection reveals that they are only an effect of a too narrow boundary. Also, if the problem that is checked would have more than one solution but only one is given, and the side moving first is trying to live, then a too narrow boundary may prevent one of the solutions and ScanLD would not find that the problem is unsuitable.
- Sometimes, the variations given in published life-and-death problems do not differ by the first move, but by a later move in the sequence. ScanLD would report any non-optimal second or later moves in a sequence as an error. Obviously this makes it necessary for a human to check whether in the publication this move is mentioned to be non-optimal too. What happens relatively frequently is that the wrong move is indicated to be wrong in the publication too but then more wrong moves appear further down in the sequence which are not recognized to be wrong. Such mistakes in books are less grave but should still not be there and are probably unintended.
- The program GoTools on which ScanLD is based, performs a search with only two possible outcomes: win or loss (apart from ko by repeating a win-loss search with increasing numbers of external ko-threats allocated to one side). A consequence is that there is no measure which could characterize one losing move as being better than another losing move. In order to come up with move sequences in which the loser plays interesting moves as well,

losing moves are chosen which postpone defeat as long as possible. Often this gives the most interesting sequence, but sometimes not.
Another consequence is that search stops for any position within the search tree when a winning move is found, not necessarily the most elegant one. An exception is the original position for which all possible first moves are investigated. This feature is intentional as an $(\alpha - \beta)$ search resulting in a number, not simply a win or loss, would take much longer.
– For large problems GoTools may simply be too slow to solve them. For the current investigations a time limit of one day was applied for evaluating the original problem position or any position that comes up after any sequence of moves in any one of the solutions.

Acknowledgements. The authors want to thank Harry Fearnley for comments on a draft version of the article. This work was made possible by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca).

References

1. Dailey, D: CGOS - A 9x9 Computer Go Server: `http://cgos.boardspace.net`
2. Gelly, Sylvain: MoGo home page: `http://www.lri.fr/~gelly/MoGo.htm`
3. Kocsis, L and Szepesvári, C: Bandit-based monte-carlo planning. ECML06, 2006.
4. Li, A and Li, Y: Korean Life-and-Death Problems (in Chinese) Beijing Sport University, ISBN 7811000601, 271 pages (2005).
5. Shen, Lei and Wolf, T: Checking Life-and-Death Problems in Go II: Results. `http://lie.math.brocku.ca/twolf/papers/bugsextra.ps`
6. Shubert, W M and Wolf, T: Tsume Go Of The Day `http://www.qmw.ac.uk/~ugah006/tsumego/index.html`
7. The KGS Go Server: `http://www.gokgs.com` and Computer Go Events: `http://www.computer-go.info/events/index.html`
8. Vesinet, J P and Wolf, T: GoTools online, `http://lie.math.brocku.ca/gotools/index.php?content=go_applet`
9. Wolf, T: Quality improvements in tsume go mass production, in the Proceedings of the Second GO & Computer Science Workshop, Cannes 1993, publ. by INRIA Sophia-Antipolis, 11 pages.
10. Wolf, T: Forward pruning and other heuristic search techniques in tsume go, Special issue of Information Sciences 122 (2000), no 1, pp. 59-76. `http://lie.math.brocku.ca/twolf/papers/jis_nat.ps`
11. Wolf, T and Pratola, M: A library of eyes in Go, II: Monolithic eyes, preprint, 16 pages, accepted for publication in the Proceedings of the International Workshop on Combinatorial Game theory at BIRS (Banff International Research Station), June 2005. `http://lie.math.brocku.ca/twolf/papers/mono.ps`

Introducing Playing Style to Computer Go

Esa A. Seuranen

Department of Electrical and Communications Engineering
Helsinki University of Technology
P.O. Box 3000, 02015 TKK, Finland
esa.seuranen@tkk.fi

Abstract. We give a brief overview of go and the methods applied in computer go. We discuss the weaknesses in the current approaches and propose a design, which is aimed to overcome some of these shortcomings. The main idea of the design is to handle the game with subgames. These subgames consist of a part of the whole board and they have a list of local purposes. The next move is chosen from the subgames according to a *playing style*, which provides values for global goals. The moves which strive towards the best outcome (in terms of global goals weighted with the style) are preferred.

1 Introduction

Go is a 4000 year old two-player board game originating from China. For the artificial-intelligence aspect go is quite interesting, because it is practically the only classical complete-information board game in which the best computer players are defeated by average human players.[1]

In this paper we discuss different aspects of go and computer go in order to state our assumptions about the game. Based on these assumptions we propose a design for a computer go player, *agent*, which should be able to overcome some weaknesses in the current approaches.

The paper is outlined as follows. In Section 2 we give an introduction to go. In Section 3 we survey the main ideas used in computer go. We introduce the concept of playing style in Section 4 and propose a design for using it in Section 5. The conclusions are given in Section 6.

2 Go

We will go over the basics briefly, that is, describe the (simplified Chinese) rules of the game, discuss the hardness of the game and emphasize the important

[1] In chess and Chinese chess (Xiang Qi) computers have reached already the level of best human players. In 2005 the Japan Shogi Association told the professional shogi players not to play against computer players publicly without a permission, so apparently the chance of defeat is quite real (`http://en.wikipedia.org/wiki/Shogi`, [14.5.2007]).

aspects of playing the game skillfully. For information about the world of go the reader is referred to [5].

A go game starts from an empty board, 19×19 intersections being the most common size. Black and white players make *moves* alternatively, i.e., place a stone of their own color on an empty intersection on the board. Placed stones remain on the board to the end of the game, unless they are captured. A set of horizontally or vertically adjacent stones of the same color, *block*, is captured if it has no adjacent empty intersections, *liberties*. A player is not allowed to place a stone on the last liberty of any of his blocks, unless he captures a group of opposing color by doing so. A move which repeats any previous board situation is illegal. These situations are referred as *kos*.[2] The game ends when both players pass, after which *dead* blocks (stones which could be captured even if they were defended) are removed. The player who has more stones and surrounded empty intersections on the board is the winner.

The rank scale of go is in Figure 1. Roughly speaking, a player is considered to be a beginner for 30–11 kyu range and an average for 11–1 kyu range. Higher dan ranks are considered to be quite strong amateur players. Professional players (indicated with *pro*) are a chapter of their own, as the strongest amateur players are considered to be similar strength to the weakest professional players. *Kami no Itte* refers to "Hand of God", i.e., perfect play. It is more or less generally believed that the difference between the best professional players and perfect play is 3–4 stones [32]. The rank difference between players gives directly the amount of handicap stones, which the weaker player should place on the board in order for both players to have equal chance of winning. For professional levels three ranks equals one stone.

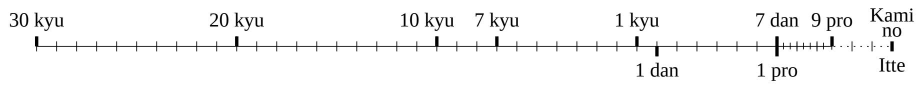


Fig. 1. Ranks in go.

In Figure 2 the rank development of European players (who have played in at least 50 tournaments and whose first rank in the system [33] was 20 kyu) is displayed over the course of years they have participated in tournaments. The triangle line is the average rank development of the players.[3] The average rank of all European tournament players is around 7 kyu.

[2] Every now and then a ko fight occurs, where players have to play threatening moves somewhere else before they can return to play a local move. The player who runs out sufficiently big threatening moves first will lose the ko fight and suffer a loss (at least locally).

[3] It is assumed that players' ranks are unchanged after their last tournament appearance.

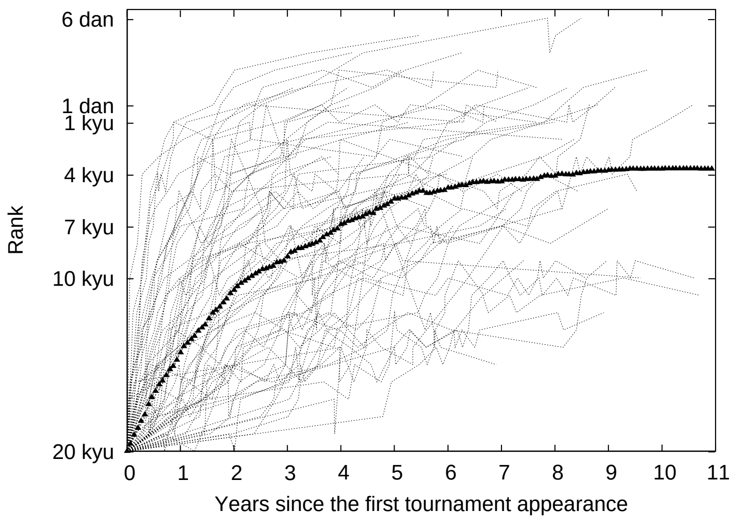


Fig. 2. Development of some human go players.

It can be said that there are four aspects in go, which make the game challenging. The first one is the difficulty of evaluating the game situation, i.e., who is winning. The second aspect is the difficulty of deciding, what should be done. The third one is about making efficient moves, i.e., finding the best local move. The fourth aspect is timing – making the right moves at the right time.

2.1 Evaluating a Game Situation

It is quite hard to name precisely the aspects that are relevant to a situation evaluation. We shall use here three terms – *territory*, *influence* and *thickness* – as the basis. The go literature supports this view, although the used terminology may vary to some degree. A situation evaluation is a combination of these three concepts.

Territory is the amount of points the player can expect to have in the end, i.e., the number of player's stones and surrounded intersections. The player's influence represents both potential of making territory and preventing his opponent from making territory. The thickness refers to the safety and stability of player's *groups* (nearby blocks of the same color). For instance, if a player has many weak groups he will have to avoid ko fights as he would surely lose something in such a fight.

2.2 Purposes of a Move

Like in any game, also in go a good move has a purpose. In fact, good moves tend to have several purposes. Such purposes include: to attack an opponent's group, to defend one's own group, to extend one's own *framework* (potential territory), to reduce the opponent's framework, to keep *sente* (initiative), to make a good *shape* (the formation of stones is efficient), to induce a bad shape for the opponent, etc. For example, "a sente move, which attacks an opponent's group in order to secure some territory while defending a weak group" sounds like a good efficient move.

These kinds of purposes represent the local aspect of the game. However, it is the global aspect of go which makes the game both complicated and intriguing. These global goals could include: to have at most one weak group at a time, to stay ahead in territory, to prevent the opponent's "sente moves, which attack my group(s) in order to secure some territory while defending his weak group", etc.

2.3 Making Efficient Moves

After the purposes for a move have been selected, it is necessary to find the best local move to accomplish the purposes – or if it seems that no move works, modifying or changing the purposes should take place. For example, saving a group might be the most important thing for the result of the game, but if there is no way to accomplish the feat currently, something else must be done.

It is quite common for stronger players to omit playing in some relatively important area if they do not find a good efficient move in the area or if they have difficulty deciding between seemingly equally good moves – they are delaying the decision on the local move further, until the game situation has changed so that the decision is easier.

2.4 Timing

The common feeling among weaker players is that the stronger player often seems to play the move the weaker player was planning to play next. And indeed, almost always the best time to play a move is the very last moment it can be played.

Figure 3 demonstrates the timing aspect. In the left diagram the starting position is shown with some possible moves for black to solidify his corner. We will have to assume that white is strong in both sides (outside the diagram). The middle diagram shows a very nice result for white, as he has managed to reduce blacks framework (moves 5 and 7) as well as leave behind the potential of making a living group in the corner with A. The right diagram shows the same moves but in the wrong order, as black has become sufficiently strong outside to resist white 5. It is a hard question, for example, when it is the correct time to play probing moves like 1 or valuable moves like A in the middle diagram.

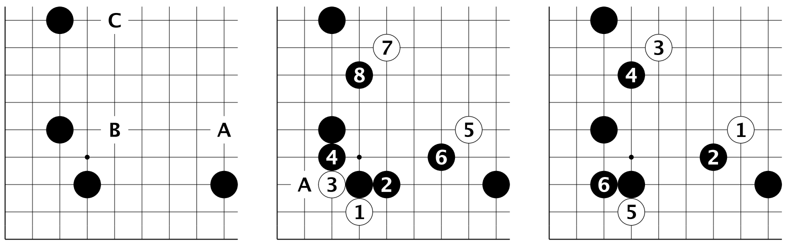


Fig. 3. A probe example.

3 Computer Go

Computer go has been studied nearly 40 years, with the current state of the art players attaining a strength around 7 kyu.[4] Here we will very briefly go through the different techniques used in constructing agents. For more thorough treatment, the reader is referred to [3, 8, 20], which cover the computer go field quite well (excluding the recent advances with Monte Carlo Go). A bibliography of go-related articles can be found in [11].

By looking at Figures 1 and 2 one can conclude that the gap between computers and the strongest humans is large. The reason for this gap is the unsuitability of the applied methods, which have been very successful with other games. We discuss evaluation functions in Section 3.1, move generators in Section 3.2 and divide-and-conquer approaches in Section 3.3.

3.1 Evaluation Function

Using an evaluation function with a tree search has been a successful approach to many games – in go the difficulty of evaluating the game situation accurately and the vast number of possible moves have prevented a breakthrough. There are four main approaches.

The first and the traditional one is a manually crafted evaluation function, which combines the estimates of territories and influence.[5] Then simply the move leading to the best game situation is chosen. The thickness aspect is usually regarded by classifying moves to urgent and non-urgent moves, and always playing an urgent move if there is one.

[4] On KGS server (`http://www.goKGS.com/`) the best agents have stabilized around 4 kyu. In Finnish ranks this would most likely correspond to 6 kyu, but since there is not that much information on agents playing in real human tournaments and people tend to play less seriously on the internet, 7 kyu is probably a reasonable estimation.

[5] Influence is usually considered to represent, more or less, the probability of a given intersection belonging to a given player – which is not quite right. In practice the territory and influence estimation is implemented by letting stones radiate power to their surroundings and summing it up. With proper radiation functions and thresholds one gets numerical estimates for both territory and influence.

Another approach is to create (manually or automatically) a rule database, which directly gives a move without any game situation evaluation [22, 24] – the advantage is the quickness and multitude of thumb rules (*proverbs*) around, which unfortunately hold only for the most of the time.

A black-box approach involves a training of a neural network to predict the winner for any game situation (for instance, see [7, 17, 23]). Neural networks do learn, but the common raw presentation of the game board along with some features has proven to be insufficient to construct a strong agent. NeuroGo [10] with its more complex board representation is the only pure neural network implementation with a moderate success.

The final approach is called Monte Carlo Go (MCG), in which simulations – i.e., random games[6] – are used to determine the move giving the best winning rate. Using an UCT algorithm [15] (running more simulations on the moves looking promising instead of running equal number of simulations for each move) and giving a proper bias for probable good moves have resulted in a very strong 9×9 agent [12]. The MCG/UCT approach is currently studied very actively in order to see, how well the approach will scale to 19×19 and how far the point of the diminishing returns is [31].[7] Based on their own experiments, the practitioners of MCG/UCT seem quite optimistic on both subjects.

3.2 Move Generators

Many methods for move generators have been already mentioned in previous sections, so there is not that much to be added here. The methods include rule-based move generators [22], neural networks [26] or goal/purpose-driven generators [34]. Usually each move generator gives a value for a move. The total value of a move on a certain intersection is a combination of the values given by different move generators.

The timing aspect of the game is not really modeled, although thinking in terms of temperature seems quite natural in this context. However, the published results (excluding the late part of the game) on the subject are scarce [6]. The temperature can be viewed as the point difference depending on which player makes a move to the area first. A point would here refer to a combined value of territory, influence and thickness resulting from the move – which makes the estimation of temperature difficult (in the endgame influence and thickness play no real role, so the estimation can be done accurately).

3.3 Divide-and-Conquer Approaches

A game of go has three phases: the opening (the board is divided into players' areas), the middle game (the areas are reduced, enlarged and exchanged) and the

[6] The game is played to the end using random moves (the legal moves are allowed, although some obviously bad moves are usually excluded).

[7] For discussions on these subjects the reader may consult the computer-go mailing list archives, `http://computer-go.org/pipermail/computer-go/`.

endgame (the exact boundaries of the areas are determined). Moderate success has been attained in opening [4,24], which is the most important part of the game – although, with amateur players the player who made the last big mistake tends to lose. The late endgame (where the board can be divided into independent subgames) can be considered to be solved [2] with combinatorial game theory [1]. The middle game has received the least attention and it is exactly the phase where the current agents are being outplayed by humans.

Besides these phases, go has some subgames as well, which are called problems. In life-and-death problems one should be able to determine the status of a group (alive, dead, depends on ko or whose turn it is) and find the correct moves to save or kill the group. In capture problems one tries to determine, whether given stones can be captured or not. And in connection problems one seeks for an answer, if a pair of stones/blocks/groups can be connected/disconnected. Several different approaches have been tried with some success, e.g., heuristics, databases, neural networks [9,16,28] and exhaustive search [14,29,30].

4 Playing Style

The current agents do not modify their way of play in accordance with their opponents or their experience. The common argument for this is that it is better to improve directly the agent against all the opponents (whereas learning has not been implemented because the current designs are not really suited for it). The argument is valid, but in our opinion a design which is capable of adjusting itself is a faster and more effective way towards a strong agent.

One could say the set of players' global goals and how he values them represent his style of play. Being strong at the local aspects of the game, tactics, is essential for a good player. But without striving towards a good style the player will not progress. Existence of styles manifests itself in the intransitive nature of the game, e.g., players A, B and C with similar strength might consistently win each other crosswise (A $>$ B, B $>$ C, but A $<$ C).

We say a player's *playing style* is a set of global features and weighted hypotheses over these features. The global features themselves are relatively simple, like "number of opponent's weak groups", "agent's own estimated territory", "number of opponent's big ko threats", etc. For instance a hypothesis "number of my weak groups $<$ number of the opponent's weak groups" could be part of a good style.

5 Tactician-Strategist Design

The nature of go is twofold: the local (tactical) battles and the global (strategic) war. If the global aspect is not handled properly, the war may be lost even if most of the local battles are won – and similarly, the war cannot be won without some prowess in the battles. Therefore we propose a two-part design consisting of a tactician and a strategist. The tactician's task is to determine the best local moves in every area (with respect to the current game situation) and their expected results as a list of accomplished purposes. The strategist then chooses the most appropriate move (with respect to the strategist's playing style) from the ones proposed by the tactician.

5.1 Tactician

Section 3.2 lists some move generators that have been and are used for finding good moves. The tactician is quite similar to a set of move generators, but the relevant difference is how the purposes of different moves are combined. The tactician constructs a subgame consisting of a part of the board and the winning condition for the subgame is a list of purposes – if and only if every purpose is fulfilled, then the subgame is won.

As there are a huge number of possible subgames, we will assume that go is an incremental game most of the time. By incremental we mean that the result of a subgame rarely changes by moves played outside the subgame, hence after each move there are not that many subgames which have to be re-evaluated. However, every now and then the result of a subgame does depend on the rest of the board. These situations are studied in [25].

The tactician has failed, if it has not proposed a move in correct area or it has estimated the result of a subgame incorrectly. High-quality game records can be used to check, if the tactician proposes a move in the correct area – if not, either the actual move in the record is bad/irrelevant or the purpose of the move is unknown to the agent. The estimation errors in the subgame results can be detected in retrospect.

For implementing the tactician we would propose to use the move generators (see Section 3.2) to provide a list of moves along with their purposes. Nearby moves are grouped together into a same subgame and the move purposes become the winning condition for that subgame. This list of purposes is pruned until the subgame can be won (or there are no purposes left). For the estimation of the subgames' result we propose the MCG/UCT approach, since it is quite suitable for binary cases (the strongest play seems to result from maximizing the winning probability instead of maximizing the winning margin), i.e., we only want to know if all the listed purposes can be accomplished or not.

5.2 Strategist

There has been some research (besides the numerous studies how to improve the evaluation function) about how one should model the global part of the game [18, 19, 27]. The lack of success with these models may result from insufficient testing or failure to model the game well. Our proposal is to incorporate playing style into the agent (see Section 4) in a form of a strategist.

The strategist should receive a list of areas and purposes, which can be fulfilled by playing in the corresponding area. The strategist then chooses the area, which gives the best result in accordance with the strategist's playing style. The weights of the style can be adjusted by training against high-quality game records: given that a proper area of play was suggested by the tactician and it was not chosen by the strategist, weights can be changed towards such values in which the correct area would have been selected. And with correct predictions the current weights can be reinforced. The hypotheses may be manually crafted, or the agent can try new random hypotheses out every now and then.

6 Conclusion

Like in any system-design process, one must know what to aim at. Obviously, we would like to construct a strong agent, but it is useful if the agent can provide accessible reasons for its moves – for the benefit of detecting deficiencies and teaching beginners. In Sections 2 and 3 we provided some background to go and computer go, while pointing out that currently the biggest bottleneck is the middle game. We proposed a design aimed at the middle game. The advantage of the design is that self-learning can be incorporated into the system. Most of the current approaches suffer from their design, in which one may expect improvements only with manual tuning or by increased computing power. In addition, several different playing styles can co-exist, making it possible to choose a style according to the opponent.

On the other hand, the proposed design needs a good representation of the board – which is a difficult problem by itself (some discussion about the subject is given in [13, 21]). Also, managing the timing aspect may prove to be very challenging, i.e., how one can include temperature into the hypotheses in an effective way. And finally, how the subgames whose results depend on some other part of the board can be handled correctly.

Acknowledgements

The author thanks the Tekniikan edistämissäätiö and the Nokia Foundation for financial support. The author is grateful to his colleagues, especially Jouni Karvo, and anonymous referees for suggestions and corrections.

References

1. E.R. Berlekamp, J.H. Conway, and R.K. Guy, *Winning Ways for You Mathematical Plays*, Academic Press, London, 1982.
2. E.R. Berlekamp, and D. Wolfe, *Mathematical Go: Chilling Gets the Last Point*, A K Peters Ltd, Natick, 1994.
3. B. Bouzy, and T. Cazenave, Computer go: an AI-oriented survey, *Artificial Intelligence Journal* **132** (2001), 39–103.
4. B. Bouzy, and G. Chaslot, Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go, In *IEEE 2005 Symposium on Computational Intelligence in Games*, G. Kendall and Simon Lucas (Eds.), Colchester, 2005, pp. 176–181.
5. R. Bozulich, *The go player's almanac 2001*, Kiseido Publishing Company, Tokyo, 2001.

6. T. Cazenave, Comparative evaluation of strategies based on the values of direct threats, In *Board Games in Academia V*, Barcelona, 2002.
7. H.W. Chan, Application of temporal difference learning and supervised learning in the game of Go, Master's thesis, Chinese University of Hong Kong, 1996.
8. K. Chen, Computer Go: Knowledge, search, and move decision, *ICGA* **24** (2001), 203–215.
9. F.A. Dahl, Honte, a Go-playing program using neural nets, 1999.
10. M. Enzenberger, Evaluation in go by a neural network using soft segmentation, In *10th Advances in Computer Games conference*, 2003, pp. 97–108.
11. M. Enzenberger, Computer Go Bibliography, [23.5.2007], `http://www.cs.ualberta.ca/~emarkus/compgo_biblio/`
12. S. Gelly, Y. Wang, R. Munos, and O. Teytaud, Modifications of UCT with patterns in Monte-Carlo Go, Tech. Report 6062, INRIA, France, 2006.
13. T. Graepel, M. Goutrié, M. Krüger, and R. Herbrich, Learning on graphs in the game of go, *Lecture Notes in Computer Science* **2130** (2001), 347–352.
14. A. Kishimoto and M. Müller, Search versus knowledge for solving life and death problems in go, In *Twentieth National Conference on Artificial Intelligence (AAAI-05)*, 2005, pp. 1374–1379.
15. L. Kocsis, and C. Szepesvári, Bandit Based Monte-Carlo Planning, In *Proceedings of the 17th European Conference on Machine Learning*, Springer-Verlag, Berlin, LNCS/LNAI 4212, 2006, pp. 282–293.
16. B. Lee, Life-and-death problem solver in go, Technical Report CITR-TR-145, University of Auckland, 2004.
17. A. Lubberts, and R. Miikkulainen, Co-Evolving a Go-Playing Neural Network, In *2001 Genetic and Evolutionary Computation Conference Workshop Program (GECCO-2001)*, San Francisco, CA: Kaufmann, 2001, pp. 14–19.
18. A.B. Meijer, and H. Koppelaar, Pursuing abstract goals in the game of Go, In *13th Belgium-Netherlands Conference on Artificial Intelligence* (BNAIC 2001), Amsterdam, 2001.
19. A.B. Meijer, and H. Koppelaar, Towards multi-objective game theory – with application to go, in *4th International Conference on Intelligent Games and Simulation*, London, EUROSIS, 2003, pp. 243–250.
20. M. Müller, Computer go, *Artificial Intelligence* **134** (2002), 145–179.
21. L. Ralaivola, L. Wu, and P. Baldi, SVM and pattern-enriched common fate graphs for the game of go, in *ESANN 2005*, Bruges, 2005, pp. 27–29.
22. S. Sei, Memory-based approach in go-program KATSUNARI, In *Complex Games Lab Workshop*, I. Frank, H. Matsubara, M. Tajima, A. Yoshikawa, R. Grimbergen, and M. Müller (Eds.), Electrotechnical Laboratory, Machine Inference Group, Tsukuba, Japan, 1998.
23. K.O. Stanley, and R. Miikkulainen, Evolving a roving eye for Go, In *Genetic and Evolutionary Computation - GECCO 2004*, New York, Springer-Verlag, 2004, pp. 1226–1238.
24. D. Stern, R. Herbrich, and T. Graepel, Bayesian pattern ranking for move prediction in the game of go, In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, ACM Press, New York, 2006, 873–880.
25. T. Thomsen, Lambda-search in game trees — with application to go, *Lecture Notes in Computer Science* **2063** (2001), 19–38.
26. E.C.D. van der Werf, J.W.H.M. Uiterwijk, E.O. Postma, and H.J. van den Herik, Local move prediction in Go, In *3rd International Conference on Computers and Games*, Edmonton, 2002.

27. S. Willmott, A. Bundy, J. Levine, and J. Richardson, Applying adversarial planning techniques to Go, *Theoretical Computer Science* **252** (2001), 45–82.
28. M.H.M. Winands, E.C.D. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk, Learning to predict life and death from go game records, In *Proceedings of JCIS 2003 7th Joint Conference on Information Sciences*, Ken Chen et al. (Eds.), Research Triangle Park, North Carolina, 2003, pp. 501–504.
29. T. Wolf, The program GoTools and its computer-generated tsume go database, In *The Game Programming Workshop in Japan '94*, M. Matsubara (Ed.), Hakone, 1994, pp. 84–96.
30. T. Wolf, Forward pruning and other heuristic search techniques in tsume go, *Information Sciences* **122** (2000), 59–76.
31. H. Yoshimoto, K. Yoshizoe, T. Kaneko, A. Kishimoto, and K. Taura, Monte Carlo has a way to go, In *Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, Menlo Park, 2006.
32. R. van Zeist, *The Magic of Go* column 157, [23.5.2007], `http://shinbo.free.fr/TheMagicOfGo/index.php?tmog=157`
33. EGF official ratings, [13.12.2006], `http://gemma.ujf.cas.cz/~cieply/GO/gor.html`
34. GNU Go, [23.5.2007], `http://www.gnu.org/software/gnugo/`

On the Parallelization of UCT

Tristan Cazenave[1] and Nicolas Jouandeau[2]

[1] Dept. Informatique cazenave@ai.univ-paris8.fr
[2] Dept. MIME n@ai.univ-paris8.fr
LIASD, Université Paris 8, 93526, Saint-Denis, France

Abstract. We present three parallel algorithms for UCT. For 9×9 Go, they all improve the results of the programs that use them against GNU GO 3.6. The simplest one, the single-run algorithm, uses very few communications and shows improvements comparable to the more complex ones. Further improvements may be possible sharing more information in the multiple-runs algorithm.

1 Introduction

Works on parallelization in games are mostly about the parallelization of the Alpha-Beta algorithm. We address here different approaches to the parallelization of the UCT algorithm.

Monte-Carlo Go has recently improved to compete with the best Go programs [3–5, 7]. We show that it can be further improved using parallelization.

Section 2 describes related work. Section 3 presents three parallel algorithms. Section 4 details experimental results. Section 5 concludes.

2 Related Works

In this section we expose related works 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.

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 games in the order of the list. At the end, the temperature is set to zero for a small number of games. After all games have been played, the value of a move is the average score of the games it has been played in first. GOBBLE-like programs have a good global sense but lack of tactical knowledge. For example, they often play useless Ataris, or try to save captured strings.

2.2 Search and Monte-Carlo Go

A very effective way to combine search with Monte-Carlo Go has been found by Rémi Coulom with his program CRAZY STONE [3]. 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 the results of the previous simulations that went through this node, and of the results of the simulations that went through its sibling nodes.

2.3 UCT

The UCT algorithm has been devised recently [6], and it has been applied with success to Monte-Carlo Go in the program MOGO [4, 5, 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 prove better). The UCT algorithm addresses the exploration/exploitation problem. UCT consists in exploring the move that maximizes $\mu_i + C \times \sqrt{log(t)/s}$. 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 t, and the number of games that start with move c_i is s.

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 parallel virtual machine that we have used to implement the parallel algorithms. Then we present in three separate subsections the three parallel algorithms.

3.1 The Parallel Virtual Machine

To improve search, we choose message passing as parallel programming model, which is implemented in the standard MPI, also supported by LAM/MPI [2]. Our virtual parallel computer, constituted with classical personal computer, sets up a fully connected network of computers. 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 two 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. All gtp read and write commands are realized from and to the master. Slaves satisfy computing requests. The maximum time taken by any slave task is specified during each computing request. Therefore, the communication time is added. According to time limits, the maximum time spent over all computing loops is defined by the sum of all slowest answers. We use a synchronous communication mode for data transmission, with time-constrained computing sequences. In the UCT context, as the algorithm is anytime, it is naturally well-adapted for synchronous programming.

```
    MASTER_PART:
singleRunParallelUCTMove($goban[\,], color, ko, time$)
1   $best \leftarrow -1$;
2   $(wins[\,], games[\,]) \leftarrow$ initialParallelUCTMove($goban[\,], color, ko, time$);
3   for $j \leftarrow 0$ to $goban.size()$
4       $best \leftarrow$ max($best, wins[j]/games[j]$);
5   return $best$;

initialParallelUCTMove($goban[\,], color, ko, time$)
1   for $i \leftarrow 0$ to $goban.size()$
2       $wins[i] \leftarrow 0$;
3       $games[i] \leftarrow 0$;
4   broadcast($goban[\,], color, ko, time$);
5   for $i \leftarrow 0$ to $nbSlaves$
6       receiveUCTSequences($newWins[\,], newGames[\,]$);
7       for $j \leftarrow 0$ to $goban.size()$
8           $wins[j] \leftarrow wins[j] + newWins[j]$;
9           $games[j] \leftarrow games[j] + newGames[j]$;
10  return $(wins[\,], games[\,])$;

    SLAVE_PART:
singleRunParallelUCTMoveSlaveLoop()
1   while($true$)
2       if(SingleQueryUCTSlaveLoop() $== END\_GAME$) break;
3   return;

SingleQueryUCTSlaveLoop()
1   if(receive($goban[\,], color, ko, time$) $== END\_GAME$) return $END\_GAME$;
2   for $i \leftarrow 0$ to $goban.size()$
3       $wins[i] \leftarrow 0$;
4       $games[i] \leftarrow 0$;
5   $(wins[\,], games[\,]) \leftarrow$ playUCTSequences($goban[\,], color, ko, time$);
6   send($wins[\,], games[\,]$);
7   return $CONTINUE$;
```

ALG. 1: Single-Run Parallel Algorithm.

3.2 Single-Run Parallelization

The single-run parallelization consists in running multiple UCT algorithms in parallel without communication between the processes. Each process has a different seed for the random-number generator, so they do not develop the same UCT tree. When the time is over, or when the maximum number of random games is reached, each slave sends back to the master the number of games and the number of wins for all the moves at the root node of the UCT tree. The master process then simply adds the number of games and the number of wins of the moves for all the slave processes.

The master part and the slave part of the single-run parallelization are given in algorithm 1.

3.3 Multiple-Runs Parallelization

On the contrary of the single-run parallelization algorithm, the multiple-runs parallelization algorithm shares information between the processes. It consists in updating the number of games and the number of wins for all the moves at the root of the shared UCT tree every fixed amount of time. The master process starts with sending the goban, the color to move, the ko intersection and the initial thinking time to the slaves, then all the slaves start computing their UCT trees, and after the initial thinking time is elapsed, they all send the number of wins and the number of games for the root moves to the master process. Then the master process adds all the results for all the moves at the root, and sends back the information to the slaves. The slaves then initiate a new UCT computation with the same shared root moves information. The communication from the slaves to the master, the update of the master root moves information, the update of the slaves root moves information and the slaves computations are then run until the overall thinking time is elapsed. It is important to notice that at line 5 of the `multipleQuerySlaveLoop` function, the `newWins` and `newGames` arrays contain the difference between the number of wins (resp. games) after the UCT search and the number of wins (resp. games) before the UCT search.

Another important detail of the algorithm is that in the slaves, the number of wins and the number of games of the root moves are divided by the number of slaves. During the experiments of the multiple-runs algorithm, we tried not to divide, and the results were worse than the non-parallel algorithm. Dividing by the number of slaves makes UCT develop its tree in the slaves in a similar way as it would without sharing information, however the average scores of the root moves are more accurate than without sharing information. The improvement comes from the improved average scores.

The master part and the slave part of the multiple-runs parallelization are given in algorithm 2.

3.4 At-the-leaves Parallelization

At-the-leaves parallelization consists in replacing the random game at a leaf of the UCT tree with multiple random games run in parallel on the slave processes. This type of parallelization costs much more in communication time than the two previous approaches since communications between the master and the slaves occur for each new leaf of the UCT tree.

In the at-the-leaves parallelization algorithm, the master is the only one to develop the UCT tree. For each new leaf of the UCT tree, it sends to the slaves the sequence that leads from the root of the tree to the leaf. Then each slave plays a pre-defined number of random games that start with the sequence, and returns the average score of these random games. The master collects all the averages of the slaves and computes the average of the averages.

```
    MASTER_PART:
multipleRunsParallelUCTMove($goban[\,], color, ko, time$)
1   $best \leftarrow -1$;
2   $(wins[\,], games[\,]) \leftarrow$ initialParallelUCTMove($goban[\,], color, ko,$
                                                   $initialPassTime$);
3   for $j \leftarrow 0$ to $goban.size()$
4       | $best \leftarrow$ max($best, wins[j]/games[j]$);
5   $time \leftarrow time - initialPassTime$;
6   while($runPassTime < time$)
7       | $(wins[\,], games[\,]) \leftarrow$ runParallelUCTMove($wins[\,], games[\,],$
                                                   $runPassTime$);
8       | $time \leftarrow time - runPassTime$;
9   for $j \leftarrow 0$ to $goban.size()$
10      | $best \leftarrow$ max($best, wins[j]/games[j]$);
11  return $best$;

runParallelUCTMove($wins[\,], games[\,], time$)
1   broadcast($wins[\,], games[\,], time$);
2   for $i \leftarrow 0$ to $nbSlaves$
3       | receiveUCTSequences($newWins, newGames$);
4       | for $j \leftarrow 0$ to $goban.size()$
5           $wins[j] \leftarrow wins[j] + newWins[j]$;
6           $games[j] \leftarrow games[j] + newGames[j]$;
7   return $(wins[\,], games[\,])$;

    SLAVE_PART:
multipleRunsParallelUCTMoveSlaveLoop()
1   while($true$)
2       | if(SingleQueryUCTSlaveLoop() $== END\_GAME$) break;
3       | $state \leftarrow CONTINUE$;
4       | while($state == CONTINUE$)
5           $state \leftarrow$ multipleQueryUCTSlaveLoop();
6   return;

multipleQueryUCTSlaveLoop()
1   if(receive($wins[\,], games[\,], time$) $== END\_LOOP$) return $END\_LOOP$;
2   for $i \leftarrow 0$ to $goban.size()$
3       | $wins[i] \leftarrow wins[i]/nbSlaves$;
4       | $games[i] \leftarrow games[i]/nbSlaves$;
5   $(newWins[\,], newGames[\,]) \leftarrow$ continueUCTSequences($time$);
6   send($newWins[\,], newGames[\,]$);
7   return $CONTINUE$;
```

ALG. 2: Multiple-Runs Parallel Algorithm.

```
    MASTER_PART:
AtLeavesParallelUCTMove($goban[\,], color, ko, time$)
1   $best \leftarrow -1$;
2   broadcast($goban[\,], color, ko$);
3   while(moreTime($time$))
4       $sequence[\,] \leftarrow$ getUCTSequence()
5       $newWins \leftarrow$ runParallelImproveAtLeaves($sequence[\,]$);
8       for $nodeId$ $in$ $sequence[\,]$
6           $wins[nodeId] \leftarrow wins[nodeId] + newWins$;
7           $games[nodeId] \leftarrow games[nodeId] + 1$;
8   for $j \leftarrow 0$ to $goban.size()$
9       $best \leftarrow$ max($best, wins[j]/games[j]$);
10  return $best$;

runParallelImproveAtLeaves($sequence[\,]$)
1   broadcast($sequence[\,]$);
2   $improvedWins \leftarrow 0$;
3   for $i \leftarrow 0$ to $nbSlaves$
4       receive($nodeWins$);
5       $improvedWins \leftarrow improvedWins + nodeWins$;
6   return $improvedWins/nbSlaves$;

    SLAVE_PART:
atLeavesParallelSlaveLoop()
1   while($true$)
2   if(receive($goban[\,], color, ko$) $==$ $END\_GAME$) break;
3       $state \leftarrow CONTINUE$;
4       while($state == CONTINUE$)
5           $state \leftarrow$ atLeavesQuerySlaveLoop();
6   return;

atLeavesQuerySlaveLoop()
1   if(receive($sequence[\,]$) $==$ $END\_LOOP$) return $END\_LOOP$;
2   for $i \leftarrow 0$ to $sequence.size()$
3       playMove($sequence[i]$);
4   $nodeWins \leftarrow 0$
5   for $i \leftarrow 0$ to $nbGamesAtLeaf$
6       $newNodeWins \leftarrow$ playRandomGame();
7       $nodeWins \leftarrow nodeWins + newNodeWins$;
8   send($nodeWins/nbGameAtLeaf$);
9   return $CONTINUE$;
```

ALG. 3: At-The-Leaves Parallel Algorithm.

The master part and the slave part of the at-the-leaves parallelization are given in algorithm 3.

4 Experimental Results

Tests are run on a simple network of computers running LINUX 2.6.18. The network includes 100 Mb switches. The BogoMips rating of each node is approximately 6000.

In our experiments, UCT uses $\mu_i + \sqrt{\frac{log(t)}{10 \times s}}$ 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 [3].

Table 1 gives the results (% of wins) of 200 9×9 games (100 with black and 100 with white) for the single-run parallel program against GNU GO 3.6 default level. The parallel algorithm has been tested with either 3,000 simulations (random games) for each UCT search, or 10,000 simulations. The single-run parallelization improves the result bringing them from 27.5% for 1 CPU and 3,000 games to 53.0% for 16 CPUs and 3,000 games per CPU. Concerning the experiments with 10,000 games per CPU, the results increase from 45.0% for 1 CPU to 66.5% for 16 CPUs. For the single-run parallelization, the communication time is very small compared to the computation time. The single-run parallelization successfully improves the UCT algorithm.

Table 1. Results of the single-run program against GNU GO 3.6.

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	27.5%	40.0%	48.0%	55.5%	53.0%
10,000 simulations	45.0%	62.0%	61.5%	65.0%	66.5%

Table 2 gives the results of 200 9×9 games for the multiple-runs parallel program against GNU GO 3.6 default level. In these experiments, the multiple-runs algorithms updates the shared information every 250 simulations. The results are similar to the results of the single-run parallelization. They are slightly better with 10,000 simulations.

Table 2. Results of the multiple-runs program against GNU GO 3.6.

	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	20.0%	32.0%	48.0%	53.5%	56.0%
10,000 simulations	49.0%	58.5%	72.0%	72.0%	68.0%

Table 3 gives the results of 200 9×9 games for the at-the-leaves parallel program against GNU GO 3.6 default level. We can see an improvement when the number of

CPUs increases from 1 to 8. However increasing the number to more than 8 does not improve much as can be seen from the second line of the table, where the slaves all play 8 games per leaf. Playing 8 games per leaf is equivalent to having 8 times more CPUs with one game per leaf.

Concerning the 10,000 simulations experiments, the percentage of wins also increases until 8 CPUs.

Table 3. Results of the at-the-leaves parallel program against GNU GO 3.6.

	nbGamesAtLeaf	1 CPU	2 CPUs	4 CPUs	8 CPUs	16 CPUs
3,000 simulations	1	21.0%	35.0%	42.0%	46.0%	45.0%
3,000 simulations	8	54.5%	48.5%	49.5%	47.5%	51.0%
10,000 simulations	1	47.0%	53.5%	53.5%	69.5%	62.0%

Table 4 shows the communication overhead for the at-the-leaves parallel program. For one CPU, the slave process runs on the same machine as the master process, so the communication time is small and the time used to find the first move on an empty 9×9 goban can be used as a reference for a non-parallel program. We see in the next columns that the communication time progressively increases, doubling the thinking time for 8 CPUs.

Table 4. Times for the first move for the at-the-leaves parallel program.

	nbGamesAtLeaf	1 CPU	2 CPUs	4 CPUs	8 CPUs
10,000 simulations	1	6.21s.	10.85s.	11.56s.	13.07s.

The communication time for the at-the-leaves parallel program is significantly higher than the two previous algorithms. Given that it gives similar improvements in level, it is preferable to use the single-run or multiple-runs algorithms. The at-the-leaves parallelization could be of interest to multiple CPUs machines with a shared memory where the communication costs are less of a problem.

5 Conclusion

We have presented three parallel algorithms that improve UCT search. They all give similar improvements. The single-run parallelization is the most simple one and also the one that uses the fewest communications between the processes.

The at-the-leaves parallelization currently costs too much communications, however it could still be interesting on a multiple CPUs machine.

We believe that the multiple-runs algorithm can be further improved to share more information and then may become the best algorithm.

References

1. Bruegmann, B.: Monte Carlo Go. ftp://ftp-igs.joyjoy.net/go/computer/mcgo.tex.z (1993)
2. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium. (1994) 379–386
3. Coulom, R.: Efficient selectivity and back-up operators in monte-carlo tree search. In: Proceedings Computers and Games 2006, Torino, Italy (2006)
4. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in Monte-Carlo go. Technical Report 6062, INRIA (2006)
5. Gelly, S., Wang, Y.: Exploration exploitation in go: UCT for Monte-Carlo go. In: NIPS-2006: On-line trading of Exploration and Exploitation Workshop, Whistler Canada (2006)
6. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: ECML-06. Number 4212 in LNCS, Springer (2006) 282–293
7. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo go. In: CIG 2007, Honolulu, USA (2007) 175–182

Monte-Carlo Go with Knowledge-Guided Simulations

Keh-Hsun Chen and Peigang Zhang

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

Abstract. We identify important Go domain knowledge which is computable in a speed comparable to random simulations in Monte-Carlo Go. We design knowledge-guided simulations to be combined with UCT algorithm for move decision in 9 × 9 Go. We demonstrate the merit of this approach through extensive testing games against three top 9 × 9 Go programs available.

1. Introduction

Monte-Carlo (MC) tree search based Go programs have achieved great success on small boards since Computer Olympiad 2006. CRAZYSTONE and MOGO have significantly outperformed all traditional knowledge and search based Go programs on 9 × 9 and 13 × 13 games. At the heart of this new generation of Monte-Carlo Go programs is the algorithm called UCT (Upper Confidence Bounds applied to Trees) [8, 7], which guides the program toward winning moves based on statistical results of random games played at selected nodes of dynamically growing MC search tree. These programs typically use very little Go knowledge.

In contrast, the first author's program GO INTELLECT is knowledge intense, yet on 9 × 9 games it was outplayed by CRAZYSTONE at the 2006 Computer Olympiad and in subsequent long testing serious. The authors are convinced that Monte-Carlo tree search is the best known approach to computer Go. Last Fall, we each started developing our own MC 9 × 9 Go programs, GO INTELLECT III 9 × 9 and JIANGO 9 × 9, respectively. We both embraced UCT algorithm and tried to incorporate Go knowledge in "random" simulations.

This paper describes the Monte-Carlo 9 × 9 Go move decision process as implemented in GO INTELLECT III 9 × 9. The emphasis is on using selected Go knowledge computable fast enough to be used in random simulations. The selected items include simple capturing, neighborhood, eye, connection, shape, and location knowledge.

We describe the basic move decision strategy in Section 2. We discuss capturing knowledge for random simulations in Section 3, neighborhood and eye knowledge in Section 4, and pattern based knowledge in Section 5. The use of Go knowledge to pre-select candidate moves at the top level is discussed in Section 6. Section 7 presents experimental results showing the merit of knowledge enhancements on the random simulations. Section 8 contains concluding remarks.

2. Basic Strategy

UCT algorithm efficiently balances exploitation of the most promising moves and exploration of the most uncertain moves. It is an excellent tool for Monte-Carlo tree search. GO INTELLECT III 9 × 9 adopts the following variant of the UCT algorithm (see Algorithm 1).

The root of the MC tree represents the current Go board configuration. We assume the MC tree is already in the memory. If not, a MC tree with just the root is created.

```
While (more simulations to do) {
   current_node = root of MC tree;
   if (the current_node has a complete children set )
         /* All legal moves from the position have been
         generated or maximum number of children
         allowed is reached */
         Advance to the child$_i$ maximizing
         $r_i$ + sqrt( log(p)/(5*$n_i$));
         /*where $r_i$ is the winning rate of move$_i$, p is
         the number of simulations passing through the
         current_node, $n_i$ is # of simulations passing
         through child$_i$ */
   else {
         Add a new child to MC tree for the move with
         the next highest urgency value;
         Play a random continuation to game-end with
         move selection probabilities proportional to
         move urgency values;
         Score the new node with the win/loss result of
         the simulation;
         Update the number of games passed by adding 1
         to all ancestors;
         Update the number of wins by adding 1 for
         every other ancestor sharing the success;
   }
}
```

Algorithm 1. A variant of the UCT algorithm used by GO INTELLECT III 9 × 9.

The main while loop can be controlled by a limit number on simulations, a time limit allocation, and/or some other criteria. After random simulations, the child of the root with the highest winning rate and exceeding a threshold of minimum number of simulation games is selected for move decision. A more detailed UCT algorithm description for the game Go can be found in [7].

Two factors determine the playing strength of a MC Go program based on UCT algorithms:

a) the quality of the simulations

b) the quantity of the simulations

The quality of the simulations helps to produce more accurate evaluations and the quantity of simulations allows MC tree to grow/see deeper. Proper balance between quality and quantity is a key to the strength of a MC program. Only domain knowledge computable in a comparable time to random-move generation and execution is beneficial to be used in the simulations.

Each move is associated with a non-negative integer urgency value, which is used in weighted random-move generation with the urgency as the weight. We use the knowledge to alter the urgencies of candidate moves, which normally have default urgency 10 (this allows us to decrease urgency to a lower integer if we want). A move with higher urgency has higher probability to be selected by the weighted random-move generator for the simulation. We call such knowledge-guided random games semi-random games. Figures 1 and 2 compare a typical semi-random game with a pure-random game. Experienced Go players can see that semi-random games are much more reasonable and will carry better feedback than pure-random games in MC simulations.

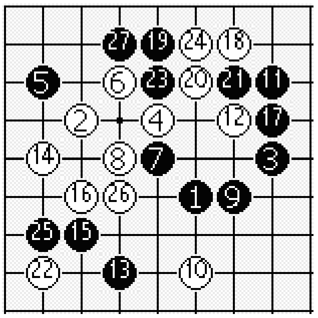


Fig. 1. A semi-random (partial) game.

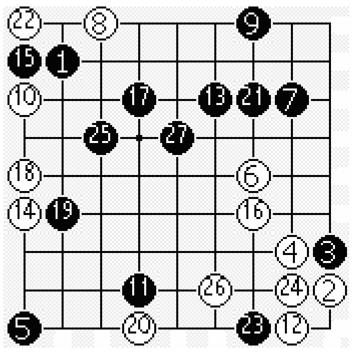


Fig. 2. A pure-random (partial) game.

We keep the MC tree in memory at all times until the game is over. When either side makes an actual move the corresponding subtree from the previous search is used for the next search. A significant amount of simulations can be reused this way.

We will discuss fast computable Go knowledge adopted by GO INTELLECT III 9 × 9 in the next three sections.

3. Capturing Knowledge

Capturing knowledge is generally recognized as extremely important for a meaningful simulation game. Coulom [6] increases urgency 1000 fold to over 10000 fold for the liberty of an ataried block. Cazenave [3] distinguishes the two liberties of a 2-liberty block by counting resulting liberties when an extra stone is played at a liberty. Play the move, called “right Atari” or “right extension”, that would generate more liberties. Unfortunately this strategy is near sighted. It does not try to know whether the “right atari” is producing an unsuccessful ladder chase. We fix this drawback by introducing pseudo ladders, which we shall discuss later in this section.

Every time a new move is played on the board during a semi-random game, the number of liberties of the block that the move is in and the numbers of liberties of its adjacent blocks are counted. If the move block has only one liberty (hence the opponent could capture it in the next move) and the block consists of more than 4 stones, undo the move – we do not allow such unreasonable moves (the chance that the block of 5 or more stones form a needed sacrifice, such as a knife-5 in life-&-death, is slim). We keep track of all the blocks adjacent to the last move that have one or two liberties. Sort out their urgencies based on whether they can be ladder captured. Although a ladder search takes only a fraction of a millisecond, it still too slow comparing to random-move generation & execution. We cannot afford the time to perform regular ladder searches during a simulation game. We generate pseudo ladders instead. We call the target of the capturing attempt the prey, and the prey's opponent the hunter. After the direction of ladder chase is determined, see Fig. 3, we scan towards that direction (the dark & light shaded squares in Fig. 3). If the scan encounters prey's support stones before hunter's stones and before Line 1, the edge of the Go board, then the prey can escape and the ladder capturing fails, otherwise it succeeds. This simple heuristic will read over 90% of the ladders correctly. The urgencies of those capture, Atari, or extend moves depend on the (pseudo) ladder capture reading outcomes.

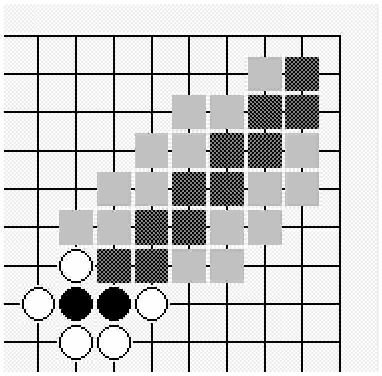


Fig. 3. Pseudo Ladder – if the directional radar scan, checking both dark and light shaded locations, picks up a black stone before it sees a white stone and before it reaches Line 1, the prey can escape from the ladder, otherwise the prey can be captured.

We classify atari moves into:

a) Critical – making the move kills the opponent block, otherwise the opponent block can escape.

b) Over-kill – the opponent block can be killed in a (pseudo) ladder even the opponent plays first.

c) False Ladder – the Atari is producing an unsuccessful ladder

d) Other

$10000 \times s^2$ is added to move urgency for a critical move where s is the number of stones of the short liberty opponent block. $100 \times s^2$ is added to the "right Atari" in category d. When the number of adjacent blocks of the prey is more than 6, the urgency of a category c atari will be reduced to 0. If not more than 6, the Atari is classified into category d. If an Atari is an over-kill, no extra urgency is added. Similar treatment is made to various types of extension-moves.

The move urgencies generated will be retained through the random game until the move is played or surrounding changes significantly. The ladder outcome can easily be affected by future moves. To resolve this problem, we keep lists of various ladder capture/escape points and dynamically update their urgencies after each move in a random simulation by redoing the pseudo ladder scans.

When a block is captured, the urgencies of the adjacent block's liberties will be reduced 100 fold (but no less than 1)

4. Neighborhood and Eyes

We update the six arrays NuEmptyNeighbors[], NuNeighbors[Black][], NuNeighbors[White][], NuEmptyDiagonals[], NuDiagonals[Black][], NuDiagonals[White][] incrementally after each move in the actual game and simulation games. Each of the six arrays uses a board point as an index, its value is self explanatory.

The information can be used to recognize solid eyes and can help check pattern conditions during semi-random games. For example, p is a solid eye for Black if

(Line[p] > 1) and (NuNeighbors[Black][p] = 4) and (NuDiagonals[Black][p] > 2) or

(Line[p] = 1) and (NuNeighbors[Black][p] = 3) and (NuDiagonals[Black] [p] = 2) or

(Pos[p] = 1) and (NuNeighbors[Black] [p] = 2) and (NuDiagonals[Black] [p] = 1)

We disallow the simulation to play at a solid eye of either color by setting its urgency to 0. For a more elaborate static eye recognition method, see [4].

5. Pattern Knowledge

We match patterns only on the surroundings of the last random move to save time. The emphases are on blocking such as in Fig. 4 left, cutting such as in Fig 4 right, extending such as in Fig. 5 left, and separating such as in Fig. 5 right. The patterns are centered around the last move and are not restricted to 3 by 3 region as in some other programs. All 8 rotations and symmetries are checked. Urgency values are updated for the move marked by a square and are applied to both sides. So if the one to play does not play there in the next move, it will have high urgency for the opponent also.

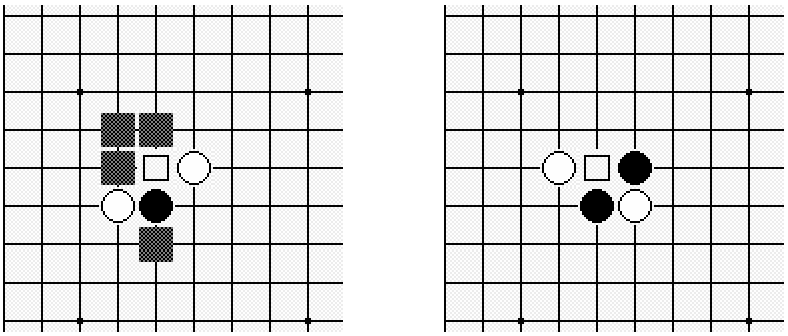


Fig. 4. The left is a blocking pattern and the right is a cutting pattern. The square marks the pattern move (currently empty). White represents ToPlay color and black represents the Opponent color. Dark shade square represents empty or Opponent.

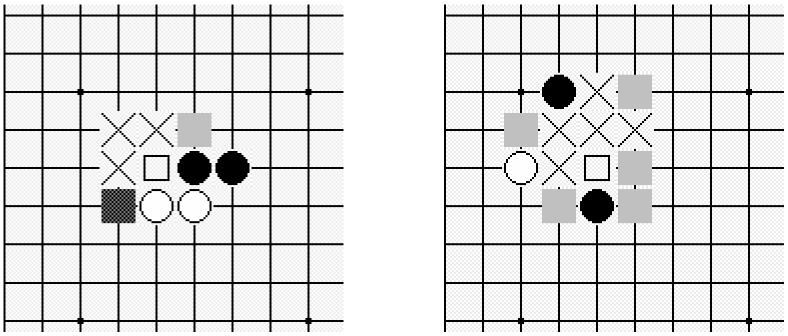


Fig. 5. The left is an extending pattern and the right is a separating pattern. The X is required to be empty. The light shade represents empty or ToPlay.

Counting liberties, reading pseudo-ladders, and matching patterns are a little more time consuming than generating random moves. As a result, only 1500 knowledge guided simulations can be performed in a second, which is one order slower than the simulation rates of those MC programs using little knowledge. But the quality of the simulations is much better providing much more accurate statistical feedback in MC tree search.

In random simulations, the default move urgency is 10. In order to reduce frequencies of Line 1 moves at early stage of a random simulation, Line 1 moves have a reduced default urgency 1 before move 45 (including moves in both the actual game and its simulation extension). The weight of a pattern move is typically in hundreds. A capture or escape move may have an urgency in tens of thousands.

6. Selection of Top Level Candidate Moves by Knowledge

GO INTELLECT III 9 × 9 uses knowledge to select one eighth of all legal moves, but no less than 4 (unless there are fewer than 4 legal moves), as the top level candidates in the MC tree. And these selected candidate moves are added to the tree in the order determined by knowledge. At the top level, restricting candidate moves to top a few best moves as judged by program knowledge has the advantage of avoiding really crazy moves and make the program more focused on exploring small number of best candidates. We shall see from the experiment result in the next section, the program versions using this strategy outperform the program versions generating all candidate moves at the top level unless the allowed number of simulations per move reaches more than a half million.

7. Experimental Results

We have performed a through test of GO INTELLECT III 9 × 9 (GI3). The testing opponents are three well established programs: GNU GO 3.6 (default version) (Gnu), 2006 Computer Olympiad 9 × 9 Go Champion CRAZYSTONE0006 (30 minutes time limit setting) (Cra), 2005 Computer Olympiad 9 × 9 Go Champion GO INTELLECT 2005 (30 minutes time limit setting) (GI05). We tested on two groups of GI3 variants. The first group does not use knowledge selection on top level moves, the second group does. Each group has 7 versions with the limit of the number of semi-random simulations per move set to 12.5K, 25K, 50K, 100K, 200K, 400K, & 800K respectively. Each version plays 40 games against each of Gnu, Cra, & GI05. On half of the games GI3 took Black, the other half of the games GI3 took White. In order to investigate the effect of the amount of simulations, GI3 is controlled by the number of semi-random simulations not by the time used. A total of 840 testing games were played on dozens of computers. Most of the machines had 3-GHz Pentium 4 processors. 12.5K versions of GI3 use about 3 minutes per game for its moves. 100K versions use about 20 minutes per game, and 800K versions average about 2 hours and 45 minutes a game.

Fig. 6 shows that the performance improves with an increasing amount of simulations. With each doubling of the simulation limit, the percentage of losing games is reduced by an average of 17%. The versions with top-level candidate moves selected by knowledge in general outperform the versions without this selection until the limit on the number of simulations allocated per move becomes real large (a half million or more).

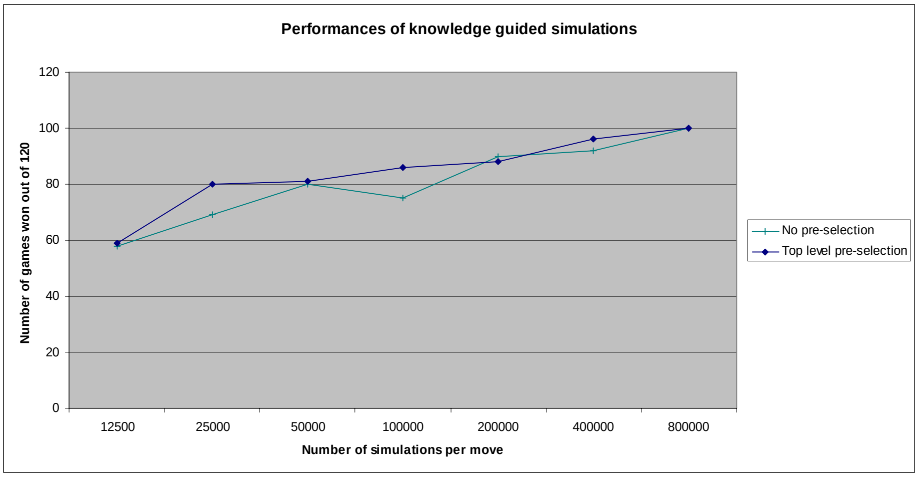


Fig. 6. Total games won by GI3 out of 120 testing games for each setting.

Fig. 7 combines the numbers of wins of the pre-select version and the non-pre-select version for each color and each simulation limit. The number of wins shown is again out of 120 total games. With 6.5 point komi, GI3 slightly prefers playing Black.

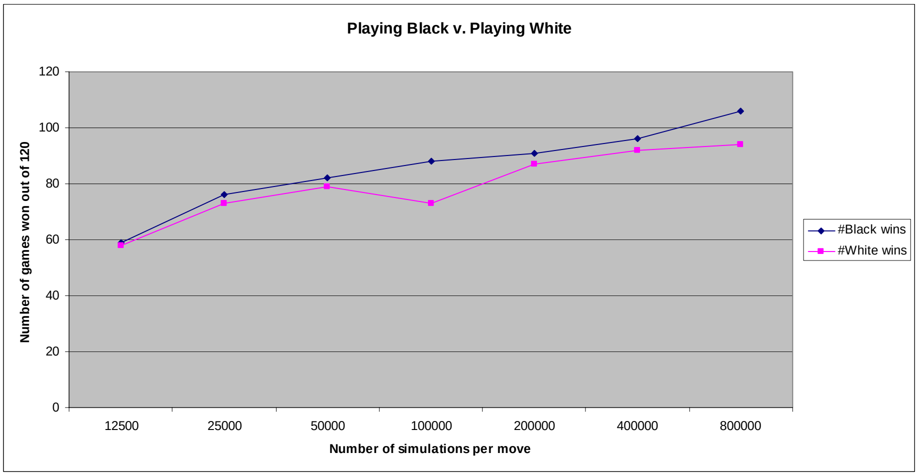


Fig. 7. This chart compares the performances of GI3 when playing Black vs. when playing White.

8. Conclusion

This paper demonstrates that improving the quality of simulations is an effective way to improve the performance of a MC Go program. If a key domain-knowledge item can be computed in a speed comparable to the speed of generating and executing random moves, then implementing it in the simulations is likely to improve the program performance. We presented a number of such knowledge items in this paper. The authors witnessed the performance improvement of our programs from adding "fast" knowledge into the simulations in the past few months. The process itself involved a lot of try-and-error adjustments. It is difficult to give an exact assessment of the contribution of each knowledge item. But the overall improvement has been very clear – the first prototype Monte-Carlo GI3 could not win more than 1/3 of the games against the three programs used in the final testing.

The UCT algorithm works well on small Go boards. But on 19 × 19 Go board, a semi-random simulation lasts several hundred moves; the simulation result becomes much more random and much less reliable. We do not believe UCT can be scaled up to 19 × 19 games directly. Integration of combinatorial game theory with Monte-Carlo tree search gives us new hope in tackling Go games on large boards.

References

1. Bouzy, B.: Associating shallow and selective global tree search with Monte Carlo for 9 × 9 Go. In van den Herik, H. J., Björnsson, Y., and Netanyahu, N. S. Editors, Forth International Conference on Computers and Games, Ramat-Gan, Israel, 2004.
2. Bouzy, B.: Move pruning techniques for Monte-Carlo Go. In Advances in Computer Games 11, Taipei, Taiwan, 2005.
3. Cazenave, T.: Playing the Right Atari Improves Monte-Carlo Go, working paper, 2006.
4. Chen, K. and Chen Z.: Static Analysis of Life and Death in the game of Go, Information Sciences, Vol. 121, Nos. 1-2, 113~134, 1999.
5. Coquelin, P. and Munos, R.: Bandit Algorithms for Tree Search, Technical Report INRIA RR-6141, 2007.
6. Coulom, R.: Efficient selectivity and backup operations in monte-carlo tree search. In Ciancarini, P., van den Herik, H. J., editors, Proceedings of the 5th International Conference on Computer and Games, Turin, Italy, 2006.
7. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with Patterns in Monte-Carlo Go. Technical Report INRIA RR-6062, 2006.
8. Kocsis, L. and Szepesvári, C.: Bandit based monte-carlo planning. European Conference on Machine Learning, pages 282-293, 2006.

Computing Elo Ratings of Move Patterns in the Game of Go

Rémi Coulom

Université Charles de Gaulle, INRIA SEQUEL, CNRS GRAPPA, Lille, France

Abstract. Move patterns are an essential method to incorporate domain knowledge into Go-playing programs. This paper presents a new Bayesian technique for supervised learning of such patterns from game records, based on a generalization of Elo ratings. Each sample move in the training data is considered as a victory of a team of pattern features. Elo ratings of individual pattern features are computed from these victories, and can be used in previously unseen positions to compute a probability distribution over legal moves. In this approach, several pattern features may be combined, without an exponential cost in the number of features. Despite a very small number of training games (652), this algorithm outperforms most previous pattern-learning algorithms, both in terms of mean log-evidence (−2.69), and prediction rate (34.9%). A 19×19 Monte-Carlo program improved with these patterns reached the level of the strongest classical programs.

1 Introduction

Many Go-playing programs use domain knowledge encoded into patterns. The kinds of patterns considered in this paper are heuristic move patterns. These are general rules, such as "it is bad to play in the corner of the board", "it is good to prevent connection of two opponent strings", "don't fill-up your own eyes", or "when in atari, extend". Such knowledge may be used to prune a search tree, order moves, or improve random simulations in Monte-Carlo programs [2, 8].

Move patterns may be built by hand, or generated automatically. A popular approach to automatically generate patterns is supervised learning [1, 4, 6, 7, 9, 12–14]: frequent patterns are extracted and evaluated from game records of strong players. In this approach, expert knowledge is used to produce a relevant encoding of patterns and pattern features, and a machine-learning algorithm evaluates them. The advantage of automatic pattern learning over hand-made patterns is that thousands of patterns may be generated and evaluated with little effort, and little domain expertise.

This paper presents a new supervised pattern-learning algorithm, based on the Bradley-Terry model. The Bradley-Terry model is the theoretical basis of the Elo rating system. The principle of Elo ratings, as applied to chess, is that each player gets a numerical strength estimation, computed from the observation of past game results. From the ratings of players, it is possible to estimate a probability distribution over the outcome of future games. The same principle can be applied to move patterns: each sample move in the training database can be considered as a victory of one pattern over the others, and can be used to compute pattern ratings. When faced with a new position, the Elo ratings of patterns can be used to compute a probability distribution over all legal moves.

1.1 Related Work

This algorithm based on the Bradley-Terry model is very similar in spirit to some recent related works, but provides significant differences and improvements.

The simplest approach to pattern learning consists in measuring the frequency of play of each pattern [4, 9]. The number of times a pattern is played is divided by the number of times it is present. This way, the strongest patterns get a higher rating because they do not stay long without being played. A major weakness of this approach is that, when a move is played, the strengths of competing patterns are not taken into consideration. In the Elo-rating analogy, this would mean estimating the strength of a player with its winning rate, regardless of the strength of opponents. By taking the strength of opponents into account, methods based on the Elo rating system can compute more accurate pattern strengths.

Stern, Herbrich, and Graepel [12] address the problem of taking the strength of opponents into account by using a model extremely similar to Elo ratings. With this model, they can compute high-quality probability distributions over legal moves. A weakness of their approach, however, is that they are restricted to using only a few move features, because the number of patterns to evaluate would grow exponentially with the number of features.

In order to solve the problem of combining move features, Araki, Yoshida, Tsuruoka, and Tsujii [1] propose a method based on maximum-entropy classification. A major drawback of their approach is its very high computational cost, which forced them to learn on a restricted subset of moves, while still taking 8.75 days of computation to learn. Also, it is not clear whether their method would provide a good probability distribution over moves, because, like the frequency-based approach, it doesn't take the strength of opponent patterns into account.

A generalized Bradley-Terry model, when combined with the minorization-maximization algorithm to compute its maximum likelihood, addresses all the shortcomings of previous approaches, by providing the algorithmic simplicity and efficiency of frequency-based pattern evaluation, with the power and theoretical soundness of methods based on Bayesian inference and maximum entropy.

1.2 Paper Outline

This paper is organized as follows: Section 2 explains the details of the theory of minorization-maximization and generalized Bradley-Terry models, Section 3 presents experimental results of pattern learning, and Section 4 describes how these patterns were applied to improve a Monte-Carlo program.

2 Minorization-Maximization and Generalized Bradley-Terry Models

This section briefly explains, independently of the problem of learning patterns in the game of Go, the theory of minorization-maximization and generalized Bradley-Terry models. It is based on Hunter's paper [11], where interested readers will find more generalizations of this model, with all the convergence proofs, references, and mathematical details.

2.1 Elo Ratings and the Bradley-Terry Model

The Bradley-Terry model allows to make predictions about the outcome of competitions between individuals. Its principle consists in evaluating the strength of each individual i by a positive numerical value γ_i. The stronger i, the higher γ_i. Predictions are made according to a formula that estimates the probability that i beats j:

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

The Elo rating of individual i is defined by $r_i = 400 \log_{10}(\gamma_i)$.

2.2 Some Generalizations of the Bradley-Terry Model

The Bradley-Terry model may be generalized to handle competitions involving more than two individuals. For n players:

$$\forall i \in \{1, \ldots, n\}, \ \ P(i \text{ wins}) = \frac{\gamma_i}{\gamma_1 + \gamma_2 + \ldots + \gamma_n} \ .$$

Another interesting generalization consists in considering not only individuals, but teams. In this generalization, the γ of a team is estimated as the product of the γ's of its members. For instance:

$$P(\text{1-2-3 wins against 4-2 and 1-5-6-7}) = \frac{\gamma_1\gamma_2\gamma_3}{\gamma_1\gamma_2\gamma_3 + \gamma_4\gamma_2 + \gamma_1\gamma_5\gamma_6\gamma_7} \ .$$

Note that the same γ may appear in more than one team. But it may not appear more than once in a team.

2.3 Relevance of Bradley-Terry Models

The choice of a Bradley-Terry model makes strong assumptions about what is being modeled, and may not be appropriate in every situation. First, a Bradley-Terry model cannot take into consideration situations where individual 1 beats individual 2 consistently, individual 2 beats individual 3 consistently, and individual 3 beats individual 1 consistently. The strengths are on a one-dimensional scale, which does not allow such cycles. Also, the generalization to teams assumes that the strength of a team is the sum (in terms of Elo ratings) of the strengths of its members. This is also a very strong assumption that may not be correct all the time.

2.4 Bayesian Inference

Bradley-Terry models, as described in the previous sections, provide a probability distribution over the outcomes of future competitions, given the strength of individuals that participate. Most of the time the exact value of parameters γ_i are unknown, and have to be estimated from the outcome of past competitions. This estimation can be done with Bayesian inference.

With $\boldsymbol{\gamma}$, the vector of parameters, and $\boldsymbol{R}$, past results, Bayes formula is:

$$P(\boldsymbol{\gamma}|\boldsymbol{R}) = \frac{P(\boldsymbol{R}|\boldsymbol{\gamma})P(\boldsymbol{\gamma})}{P(\boldsymbol{R})} \quad .$$

It gives a likelihood distribution over $\boldsymbol{\gamma}$, from $P(\boldsymbol{R}|\boldsymbol{\gamma})$, that is to say the Bradley-Terry model described in the previous sections, $P(\boldsymbol{\gamma})$, a prior distribution over parameters, and $P(\boldsymbol{R})$, a normalizing constant. Parameters $\boldsymbol{\gamma}$ may be estimated by finding $\boldsymbol{\gamma}^*$ that maximizes $P(\boldsymbol{\gamma}|\boldsymbol{R})$.

This optimization can be made more convenient by choosing a prior that has the same form as the Bradley-Terry model itself. That is to say, virtual results $\boldsymbol{R}'$ will serve as a prior: $P(\boldsymbol{\gamma}) = P(\boldsymbol{R}'|\boldsymbol{\gamma})$. This way, the estimation of parameters of the model will consist in maximizing $P(\boldsymbol{R}, \boldsymbol{R}'|\boldsymbol{\gamma})$.

2.5 A Minorization-Maximization Algorithm

Notations. $\gamma_1, \ldots, \gamma_n$ are the strength parameters of n individuals. N results $R_1, \ldots, R_N$ of independent competitions between these individuals are known. These competitions are of the most general type, as described in Section 2.2. The probability of one competition result may be written as

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

where A_{ij}, B_{ij}, C_{ij}, and D_{ij} are factors that do not depend on γ_i. With this notation, each $P(R_j)$ can be written in n different ways, each time as a function of one particular γ_i. E_j is defined as $E_j = C_{ij}\gamma_i + D_{ij}$, and $W_i = |\{j|A_{ij} \neq 0\}|$ is the number of wins of individual i. The objective is to maximize:

$$L = \prod_{j=1}^{N} P(R_j)$$

Derivation of the Minorization-Maximization Formula. (Readers who do not wish to understand all the details may safely skip to the formula)

Minorization-maximization is an iterative algorithm to maximize L. Its principle is illustrated on Figure 1. Starting from an initial guess $\boldsymbol{\gamma}^0$ for $\boldsymbol{\gamma}$, a function m is built, that *minorizes* L at $\boldsymbol{\gamma}^0$. That is to say, $m(\boldsymbol{\gamma}^0) = L(\boldsymbol{\gamma}^0)$, and, for all $\boldsymbol{\gamma}$, $m(\boldsymbol{\gamma}) \leq L(\boldsymbol{\gamma})$. The maximum $\boldsymbol{\gamma}^1$ of m is then computed. Thanks to the minorization property, $\boldsymbol{\gamma}^1$ is an improvement over $\boldsymbol{\gamma}^0$. The trick is to build m so

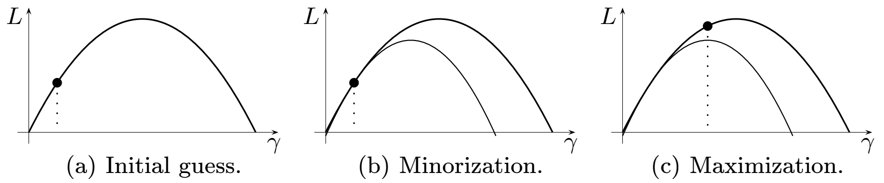


Fig. 1. Minorization-maximization.

that its maximum can be computed in closed form. This optimization algorithm is often much more efficient than traditional gradient-ascent methods.

$$L = \prod_{j=1}^{N} \frac{A_{ij}\gamma_i + B_{ij}}{C_{ij}\gamma_i + D_{ij}}$$

is the function to be maximized. L can be considered as a function of γ_i, and its logarithm is:

$$\log L(\gamma_i) = \sum_{j=1}^{N} \log(A_{ij}\gamma_i + B_{ij}) - \sum_{j=1}^{N} \log(C_{ij}\gamma_i + D_{ij}) \enspace .$$

Terms that do not depend on γ_i can be removed, and, since either $B_{ij} = 0$ or $A_{ij} = 0$, the function to be maximized becomes:

$$f(x) = W_i \log x - \sum_{j=1}^{N} \log(C_{ij}x + D_{ij}) \enspace .$$

The logarithms in the right-hand part may be minorized by their tangent at $x = \gamma_i$, as shown on Figure 2. After removing the terms that do not depend on

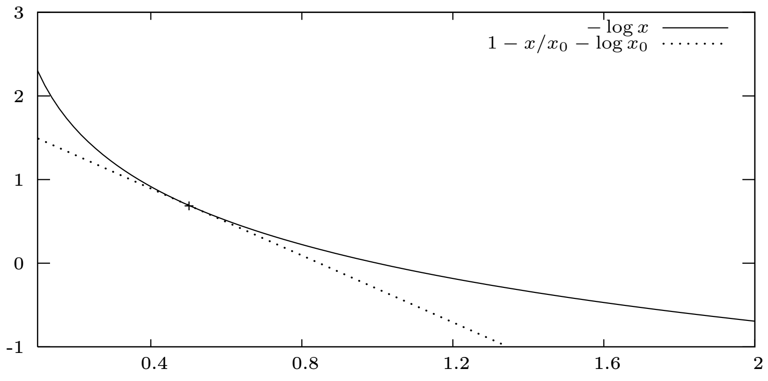


Fig. 2. Minorization of $-\log x$ at $x_0 = 0.5$ by its tangent.

x, the minorizing function to be maximized becomes

$$m(x) = W_i \log x - \sum_{j=1}^{N} \frac{C_{ij} x}{E_j} \ .$$

The maximum of $m(x)$ is at

$$x = \frac{W_i}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}} \ .$$

Minorization-Maximization Formula. So, minorization-maximization consists in iteratively updating one parameter γ_i according to this formula:

$$\gamma_i \leftarrow \frac{W_i}{\sum_{j=1}^{N} \frac{C_{ij}}{E_j}} \ .$$

If all the parameters are initialized to 1, and the number of participants in each competition is the same, the first iteration of minorization-maximization computes the winning frequency of each individual. So, in some way, minorization-maximization provides a Bayesian justification of frequency-based pattern evaluation. But running more than one iteration improves parameters further.

When players have different strengths, C_{ij} indicates the strength of team mates of i during competition j, and E_j is the overall strength of participants. With the minorization-maximization formula, a win counts all the more as team mates are weak, and opposition is strong.

Batch Updates. The minorization-maximization formula describes how to update just one γ_i. It is possible to iteratively update all the γ_i one by one, but it may be inefficient. Another possibility is to perform batch updates. A set of mutually exclusive γ_i's may be updated in one single pass over the data. Mutually exclusive means that they cannot be members of the same team. The batch-update approach still has good convergence properties [11], and offers the opportunity to re-use computations. In particular, $1/E_j$ can be computed only once in a batch.

3 Pattern-Learning Experiments in the Game of Go

A generalized Bradley-Terry model can be applied to supervised learning of Go patterns, by considering that each sample move is a competition, whose winner is the move in question, and losers are the other legal moves. Each move can be considered as a "team" of features, thus allowing to combine a large number of such features without a very high cost.

3.1 Data

Learning was performed on game records played by strong players on KGS. These game records were downloaded from the web site of Kombilo [10]. The training set was made of the 652 games with no handicap of January, 2006 (131,939 moves). The test set was made of the 551 games with no handicap of February, 2006 (115,832 moves). The level of play in these games may not be as high as the professional records used in previous research on pattern learning, but they have the advantage of being publicly available for free, and their level is more than high enough for the current level of Go-playing programs.

3.2 Features

The learning algorithm used 8 tactical features: pass, capture, extension, self-atari, atari, distance to border, distance to the previous move, and distance to the move before the previous move. Some of these features may take more than one value, as explained in Table 1.

The 9th feature was Monte-Carlo owner. It was computed by running 63 random games from the current position. For each point of the board, the number of final positions owned by the player to move was counted.

The 10th feature was shape patterns. Nested circles of radius 3 to 10 according to the distance defined in Table 1 are considered, similarly to [12]. 16,780 shapes were harvested from the training set, by keeping those that appear at least 5,000 times.

Each value that these features can take is considered as a separate "individual", and is associated to one strength parameter γ_i. Since values within one feature are mutually exclusive, they were all updated together within one iteration of the minorization-maximization algorithm.

3.3 Prior

The prior was set by adding, for each γ_i, one virtual win, and one virtual loss, against a virtual opponent whose γ is 1. In the Elo-rating scale, this produces a symmetric probability distribution, with mean 0 and standard deviation 302.

3.4 Results

Table 1 lists the values of γ for all non-shape features.

Figure 3 plots the mean log-evidence per stage of the game, against the data of Stern, Herbrich, and Graepel [12]. This mean log-evidence is the mean logarithm of the probability of selecting the target move according to the Bradley-Terry model, measured over the test set. The overall mean log-evidence is -2.69, which corresponds to an average probability of 1/14.7. Uniform probability gives a mean log-evidence of -5.49, which corresponds to an average probability of 1/243.

Figure 4 is a plot of the cumulative distribution of the probability of finding the target move at a given rank, measured over the test set, and compared with other authors.

Feature	Level	γ	Description
Pass	1	0.17	Previous move is not a pass
	2	24.37	Previous move is a pass
Capture	1	30.68	String contiguous to new string in atari
	2	0.53	Re-capture previous move
	3	2.88	Prevent connection to previous move
	4	3.43	String not in a ladder
	5	0.30	String in a ladder
Extension	1	11.37	New atari, not in a ladder
	2	0.70	New atari, in a ladder
Self-atari	1	0.06	
Atari	1	1.58	Ladder atari
	2	10.24	Atari when there is a ko
	3	1.70	Other atari
Distance to border	1	0.89	
	2	1.49	
	3	1.75	
	4	1.28	
Distance to previous move	2	4.32	$d(\delta x, \delta y) = \lvert\delta x\rvert + \lvert\delta y\rvert + \max(\lvert\delta x\rvert, \lvert\delta y\rvert)$
	3	2.84	
	4	2.22	
	5	1.58	
	...	...	
	16	0.33	
	≥ 17	0.21	
Distance to the move before the previous move	2	3.08	
	3	2.38	
	4	2.27	
	5	1.68	
	...	...	
	16	0.66	
	≥ 17	0.70	
MC Owner	1	0.04	$0 - 7$
	2	1.02	$8 - 15$
	3	2.41	$16 - 23$
	4	1.41	$24 - 31$
	5	0.72	$32 - 39$
	6	0.65	$40 - 47$
	7	0.68	$48 - 55$
	8	0.13	$56 - 63$

Table 1. Model parameters for non-shape features. Each feature describes a property of a candidate move in the current position. A feature my either be absent, or take one of the values indicated in the Level column.

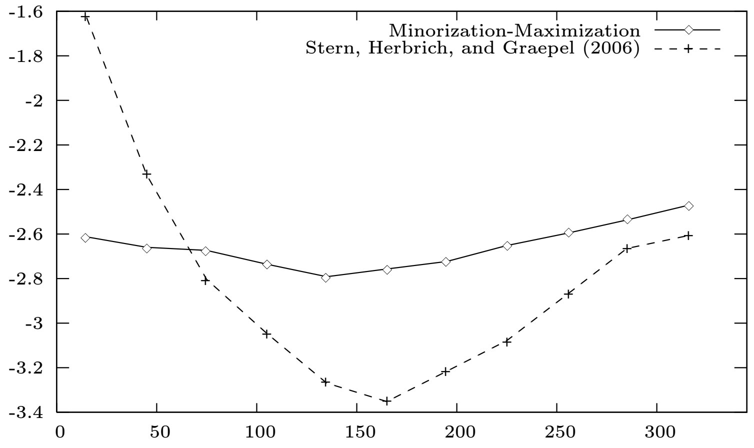


Fig. 3. Mean log-evidence per stage of the game (each point is an average over an interval of 30 moves).

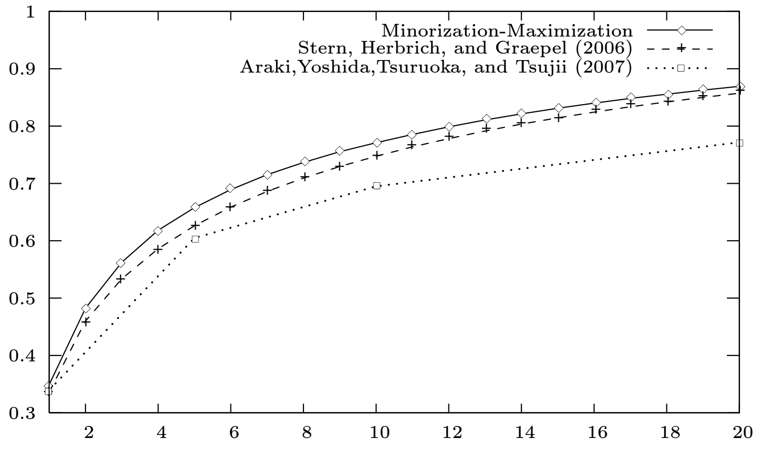


Fig. 4. Cumulative distribution: probability of finding the target move within the n best estimated moves.

3.5 Discussion

The prediction rate obtained with minorization-maximization and the Bradley-Terry model is the best among those published in academic papers. De Groot[9] claims a 42% prediction rate, so his results are still significantly better.

Despite the similarity of the cumulative distributions, the mean log-evidence per stage of the game has a very different shape from that of Stern, Herbrich, and Graepel. Their algorithm provides much better predictions in the beginning of the game, and much worse in the middle. It is worth noting also that their learning experiments used many more games (181,000 instead of 652) and shape patterns (12,000,000 instead of 16,780). So they tend to learn standard opening sequences by rote, whereas our algorithm learns more general rules.

The learning process of our algorithm is not particularly optimized, and took about one hour of CPU time and 600 Mb of RAM to complete. So it is very likely that prediction performance could be improved very easily by using more games, and more shape patterns. Most of the computation time was taken by running the Monte-Carlo simulations. In order to learn over many more games, the slow features could be trained afterward, over a small set of games.

4 Usage of Patterns in a Monte-Carlo Program

Despite the clever features of this pattern-learning system, selecting the move with the highest probability still produces a terribly weak Go player. It plays some good-looking moves, but also makes huge blunders because it really does not "understand" the position. Nevertheless, the domain knowledge contained in patterns is very precious to improve a Monte-Carlo program, by providing a good probability distribution for random games, and by helping to shape the search tree. This section briefly describes how patterns are used in Crazy Stone [5].

4.1 Random Simulations

The pattern system described in this paper produces a probability distribution over legal moves, so it is a perfect candidate for random move selection in Monte-Carlo simulations. Monte-Carlo simulations have to be very fast, so the full set of features that was described before is much too slow. Only light-weight features are kept in the learning system: 3x3 shapes, extension (without ladder knowledge), capture (without ladder knowledge), self-atari, and contiguity to the previous move. Contiguity to the previous move is a very strong feature ($\gamma = 11$), and tends to produce sequences of contiguous moves like in Mogo [8].

4.2 Progressive Widening of the Monte-Carlo Search Tree

Crazy Stone also uses patterns to prune the search tree. This is performed at a much slower rate, so the full power of complex features can be used. When a node in the Monte-Carlo search tree is created, it is searched for a while without any pruning, selecting the move according the policy of random simulations. As soon as a number of simulations is equal to the number of points of the board, this node is promoted to internal node, and pruning is applied. Pruning consists in restricting the search to the n best moves according to patterns, with n growing like the logarithm of the number of random simulations. More precisely, the nth move is added when t_n simulations have been run, with $t_1 = 0$ and $t_{n+1} = t_n + 40 \times 1.4^n$. On 19×19, thanks to the distance-to-the-previous-move feature, progressive widening tends to produce a local search, like in Mogo [8].

4.3 Performance against GNU Go

Table 2 summarizes Crazy Stone's performance against GNU Go 3.6, on an AMD Opteron at 2.2 GHz, running on one CPU. Crazy Stone ran, per second, from the empty position, 15,500 simulations on 9×9, and 3,700 on 19×19.

Pat.	P.W.	Size	Min./game	GNU Level	Komi	Games	Win ratio
-	-	9×9	1.5	10	6.5	170	38.2%
x	-	9×9	1.5	10	6.5	170	68.2%
x	x	9×9	1.5	10	6.5	170	90.6%
-	-	19×19	32	8	6.5	192	0.0%
x	-	19×19	32	8	6.5	192	0.0%
x	x	19×19	32	8	6.5	192	37.5%
x	x	19×19	128	8	6.5	192	57.1%

Table 2. Match results. P.W. = progressive widening. Pat. = patterns in simulations.

5 Conclusion

The research presented in this paper demonstrates that a generalized Bradley-Terry model is a very powerful technique for pattern learning in the game of Go. It is simple and efficient, can combine several features, and produces a probability distribution over legal moves. It is an ideal tool to incorporate domain knowledge into Monte-Carlo tree search.

Experiment results clearly indicate that significant progress can be made by learning shapes over a larger amount of training games, and improving features. In particular, the principle of Monte-Carlo features is very powerful, and could be exploited more, as Bouzy did with history and territory heuristics [3].

Also, the validity of the model could be tested and improved. First, using all the moves of one game as sample data breaks the hypothesis of independence between samples, since consecutive positions are very similar. Sampling one or two positions per game might be better. Also, the linearity hypothesis of the generalized Bradley-Terry model, according to which the strength of a team is the sum of the strengths of its members, is likely to be wrong. Estimating the strength of some frequent feature pairs separately might improve predictions.

Acknowledgments

I thank David Stern, Ralf Herbrich and Thore Graepel for kindly providing files with their performance data, so that I could plot mine against theirs. I am also grateful to the workshop reviewer and the readers of the computer-go mailing list for their comments that helped to improve the first version of this paper.

References

1. Nobuo Araki, Kazuhiro Yoshida, Yoshimasa Tsuruoka, and Jun'ichi Tsujii. Move prediction in Go with the maximum entropy method. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games*, 2007.
2. Bruno Bouzy. Associating domain-dependent knowledge and Monte-Carlo approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, 175(4):247–257, November 2005.
3. Bruno Bouzy. History and territory heuristics for Monte-Carlo Go. *New Mathematics and Natural Computation*, 2(2):1–8, 2006.
4. Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of K-nearest-neighbor patterns for 19x19 Go. In G. Kendall and Simon Lucas, editors, *IEEE Symposium on Computational Intelligence in Games*, pages 176–181, Colchester, UK, 2005.
5. Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In P. Ciancarini and H. J. van den Herik, editors, *Proceedings of the 5th International Conference on Computer and Games*, Turin, Italy, 2006.
6. Fredrik A. Dahl. Honte, a Go-playing program using neural nets. In Johannes Fürnkranz and Miroslav Kubat, editors, *16th International Conference on Machine Learning, Workshop Notes: Machine Learning in Game Playing*, Bled, Slovenia, 1999.
7. Herbert Enderton. The Golem Go program. Technical Report CMU-CS-92-101, School of Computer Science, Carnegie-Mellon University, 1991.
8. Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report RR-6062, INRIA, 2006.
9. Franck de Groot. Moyo Go Studio. http://www.moyogo.com/, 2007.
10. Ulrich Görtz and William Shubert. Game records in SGF format. http://www.u-go.net/gamerecords/, 2007.
11. David R. Hunter. MM algorithms for generalized Bradley-Terry models. *The Annals of Statistics*, 32(1):384–406, 2004.
12. David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of Go. In *Proceedings of the 23rd international conference on Machine learning*, pages 873–880, Pittsburgh, Pennsylvania, USA, 2006.
13. David Stoutamire. Machine learning, game play, and Go. Technical Report TR 91-128, Center for Automation and Intelligent Systems Research, Case Western Reserve University, 1991.
14. Erik van der Werf, Jos Uiterwijk, Eric Postma, and Jaap van den Herik. Local move prediction in Go. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games, Third International Conference, CG 2002*, pages 393–412. Springer Verlag, 2003.

Grouping Nodes for Monte-Carlo Tree Search

Jahn-Takeshi Saito, Mark H.M. Winands,
Jos W.H.M. Uiterwijk, and H. Jaap van den Herik

MICC-IKAT
Universiteit Maastricht
P.O. Box 616, 6200 MD Maastricht
The Netherlands
{j.saito,m.winands,uiterwijk,herik}@micc.unimaas.nl

Abstract. Recently, Monte-Carlo Tree Search (MCTS) has substantially contributed to the field of computer Go. So far, in standard MCTS there is only one type of node: every node of the tree represents a single move. Instead of maintaining only this type of node, we propose a second type of node representing groups of moves. Thus, the tree may contain move nodes and group nodes. This article documents how such group nodes can be utilized for including domain knowledge to MCTS. Furthermore, we present a technique, called Alternating-Layer UCT, for managing move nodes and group nodes in a tree with alternating layers of move nodes and group nodes. A self-play experiment demonstrates that group nodes can improve the playing strength of a MCTS program.

1 Introduction

In the last fifteen years, Monte-Carlo methods have led to strong computer Go programs. The short history of Monte-Carlo based Go programs underwent two phases. In the first phase, Monte-Carlo was introduced [2, 3] as an evaluation function. A Monte-Carlo evaluation simply estimates the value of a game state S by statistically analyzing random games starting from S. In the second phase the focus of research shifted to Monte-Carlo Tree Search (MCTS, [4, 7, 8]).

Until now, all existing work on MCTS employs tree nodes to represent only a single move. The contribution of this work is to introduce nodes representing groups of moves to MCTS. This article presents a technique extending MCTS for managing group nodes. It enables the application of domain knowledge in MCTS in a natural way.

The remainder of this article is organized as follows. Section 2 explains MCTS and a specific move-selection function for MCTS called UCT. Section 3 introduces the concept of group nodes for MCTS and Alternating-Layer UCT. Section 4 describes an experiment comparing standard UCT and Alternating-Layer UCT and discusses the results. Finally, Section 5 gives a conclusion and an outlook to future research.

2 Monte-Carlo Tree Search

This section first describes Monte-Carlo Tree Search (MCTS) in general (Subsection 2.1). In Subsection 2.2 a specific move-selection function for MCTS, called UCT, is explained.

2.1 The Monte-Carlo Tree Search Framework

MCTS constitutes a further development of the Monte-Carlo evaluation. It provides a tree-search framework for employing Monte-Carlo evaluations at the leaf nodes of a particular search tree.

MCTS constitutes a family of tree-search algorithms applicable to the domain of board games [4, 7, 8]. In general, MCTS repeatedly applies a best-first search at the top level. Monte-Carlo sampling is used as an evaluation function at leaf nodes. The results of previous Monte-Carlo evaluations are used for developing the search tree. MCTS consists of four stages [5]. During each iteration, four stages are consecutively applied:

(1) *move-selection;*
(2) *expansion;*
(3) *leaf-node evaluation;*
(4) *back-propagation.*

Each node N in the tree contains at least three different tokens of information: (i) a move representing the game-state transition associated with this node, (ii) the number $t(N_i)$ of times the node has been played during all previous iterations, and (iii) a value $v(N)$ representing an estimate of the node's game value. The search tree is held in memory. Before the first iteration, the tree consists only of the root node. While applying the four stages successively in each iteration, the tree grows gradually. The four stages of the iteration work as follows.

(1) The move selection determines a path from the root to a leaf node. This path is gradually developed. At each node, starting with the root node, the best successor node is selected by applying a move-selection function to all child nodes. Then, the same procedure is applied to the selected child node. This procedure is repeated until a leaf node is reached.
(2) After the leaf node is reached, it is decided whether this node will be expanded by storing some of its children in memory. The simplest rule, proposed by Coulom [7], is to expand one node per evaluation. The node expanded corresponds to the first position encountered that was not stored yet.
(3) A Monte-Carlo evaluation (also called *playout* or *simulation*) is applied to the leaf node. Monte-Carlo evaluation is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves. The main idea is to play more reasonable moves by using patterns, capture considerations, and proximity to the last move.

(4) During the back-propagation stage, the result of the leaf node is back-propagated through the path created in the move-selection stage. For each node in the path back to the root, the node's game values are updated according to the updating function.[1] After the root node has been updated, this stage and the iteration are completed.

As a consequence of altering the values of the nodes on the path, the move selection of the next iteration is influenced. The various MCTS algorithms proposed in the literature differ in their move-selection functions and update functions. The following subsection briefly describes a specific move-selection function called UCT.

2.2 UCT

Kocsis and Szepesvári [8] introduced the move-selection function UCT. It was fruitfully applied in top-level Go programs such as Crazy Stone, Mogo, and Mango. These programs entered in the loop of various KGS tournaments successfully.

Given a node N with children N_i, the move-selection function of UCT chooses the child node N_i which maximizes[2] the following criterion.

$$\overline{X}_i \; + \; C \sqrt{\frac{ln \;\; t(N)}{t(N)_i}} \quad . \tag{1}$$

This criterion takes into account the number of times $t(N)$ that node N was played in previous iterations, the number of times $t(N_i)$ the child N_i was selected in previous iterations, and the average evaluation $\overline{X}_i$ of the child node N_i.

The weighted square-root term in Equation 1 describes an upper confidence bound for the average game value. This value is assumed to be normally distributed among the evaluations selected during all iterations. Each iteration passing through N represents a random experiment influencing the estimate of the mean parameter $\overline{X}$ of this distribution.

The constant C controls the balance between exploration and exploitation [5]. It prescribes how often a move represented by child node N_i with high confidence (large $t(N_i)$) and good average game value (large $\overline{X}_i$) is preferred to a move with lower confidence or lower average game value.

The update function used together with UCT sets the value $\overline{X}$ of a node N to the average of all the values of the previously selected children including the latest selected child's value $\overline{X}_i$.

3 Grouping Nodes

We describe the concept of group nodes in this section. Subsection 3.1 outlines related work in which domain knowledge is applied to form groups of moves.

[1] Coulom [7] refers to update function as back-propagation operator.
[2] In Min nodes the roles are reversed and the criterion is minimized.

Subsection 3.2 provides a detailed account of the features of group nodes. Subsection 3.3 presents an extension of the UCT which is able to manage group nodes.

3.1 Related Work

The idea of categorizing moves in games is not new. The work of Tsuruoka *et al.* [10], to name only one example, applies domain knowledge in the game of Shogi to categorize follow-up moves for a given game state according to simple features. Depending on which categories the move falls into, the amount of resources (mainly search depth) allocated to searching this particular move is fixed. In the domain of computer Go, simple features have been applied for various tasks related to search. Features have been used to assign probabilities to playing certain moves in random games or to eliminate unpromising moves.

The concept of *Metapositions* was established by Sakuta [9] to represent sets of possible game states. Metapositions are employed as nodes in game-tree search. Moreover, Metapositions were found to be suitable to represent sets of game states consistent with observations made in the imperfect information game Kriegspiel [6].

Set Pruning, introduced by Bouzy [1], is a technique, first developed for computer Go. It brings together two items: first, the idea of grouping moves according features, and second, the aspect using sets of moves in a search tree as found in Metapositions. In Set Pruning two categories of moves are maintained: moves labelled as *Good* moves and moves labelled as *Bad* moves. Then, Monte-Carlo evaluation is applied to moves in both sets and a common statistic is maintained for all moves belonging to the same category. The technique proposed here can be viewed as a twofold extension of this technique. The first extension effects the number of move groups which we suggest to generalize to more than only two. The second extension this article proposes to go beyond Set Pruning concerns the framework groups are used in: instead of considering only Monte-Carlo evaluation, we propose to apply groups to MCTS. This second extension implies that the statistics are no longer kept in only one place (a node to be evaluated). Instead, the statistics alter many nodes in the search tree. Thus, a mechanism is required to adopt the idea of groups of moves to MCTS. This mechanism is facilitated by group nodes which are the subject of the following subsection.

3.2 Group Nodes

In plain MCTS, all nodes of the tree represent simple moves. We call them *move nodes*. To facilitate the use of domain knowledge we introduce the concept of *group nodes*. A group node represents a group of moves. Domain knowledge in the form of features is used to group the nodes.

Given a move represented by a node N, we suggest to partition the set of its child nodes N_i. We call each partition a group. The partitioning is achieved by assigning each N_i to exactly one group according to whether they meet certain pairwise exclusive features.

Group nodes can function in a MCTS framework only if they provide the basic features MCTS requires a node to have (cf. Subsection 2.1). Thus, for each group node G three features are recorded: (i) a move representing the game state transition associated with this move, (ii) the number $t(G)$ of times it was visited in previous iterations, and (iii) a game value $\overline{X}_G$. Since a group node does not represent a single move, the game state is not altered whenever a group node is met during an iteration. The game value of a group node is set to represent the average game value of all nodes belonging to the group.

3.3 Alternating-Layer UCT

Alternating-Layer UCT is a technique which manages group nodes in the UCT framework [8]. The Alternating-Layer UCT maintains two types of nodes in the MCTS tree in parallel: (1) move nodes, and (2) group nodes.

When a move node N is expanded, it is expanded in two steps. First, N is expanded into group nodes (first layer). Second, each of the newly expanded group nodes is expanded into move nodes (second layer).

In the first step, all successor nodes are grouped according to features (cf. Subsection 3.1). Each group is represented by a group node G_i. These newly created group nodes become the new children of N. In the second step, each G_i is expanded. For each member move of G_i a new move node G_{ij} is created as child node of G_i.

After completing the two steps of expanding a move node, all new leaf nodes are move nodes. Because the root is a move node, all leaf nodes are move nodes at the end of each iteration. Furthermore, the structure of the search tree is such that move nodes and group nodes form alternating layers (leading to the proposed naming).

In standard implementations of the UCT algorithm, all follow-up moves of a leaf node L are chosen randomly until L has been played a critical number x of times ($t(L) > x$). Analogously, we suggest choosing the follow-up move of a leaf node L among its succeeding groups with equal probability while $t(L) > x$. This equidistribution results in an implicit weighting of moves, because the number of members may vary between the groups. We consider, e.g., that two groups G_1 and G_2 are given with n_1 or n_2 moves, respectively. Furthermore, $n_1 \gg n_2$. If both groups are selected equally often, a move which is member of G_1 is less likely to be selected than a move which is member of G_2.

4 Experiment

In this section, we test the increase of playing strength gained by Alternating-Layer UCT. Subsection 4.1 describes the setup of a self-play experiment. Subsection 4.2 presents and comments the results obtained.

4.1 Set-up

Standard UCT and the Alternating-Layer UCT were implemented for the domain of computer Go. We refer to the resulting programs as *STD* and *AL*, respectively.

Both implementations were compared in a series of 1,000 games of direct play on 9×9 boards with 5.5 points Komi (500 games for each program playing Black, respectively). The time setting was one second per move. The Monte-Carlo sampling used in both implementations is based on small hand-tuned patterns and reaches a speed of about 22,000 sampled games per second on the given hardware (cf. below). The C parameter (cf. Equation 1) was set to 1.9, determined by trial-and-error, for *STD* and *AL*. The x parameter (cf. 3.3) for describing the threshold for expanding moves was set to the number of children of the node to be expanded.

The number of iterations available per move for *STD* is 30,000. To compensate for the grouping overhead of alternating-layer UCT, *AL* was allowed only 10,000 iterations per move.

Two features were used for grouping in *AL* resulting in three types of group nodes. The first feature is proximity to the last move. The proximity chosen is the Manhattan distance of 2. (For the empty board, the last move is set to be the center.) The second feature determines whether a move is a border move. The three resulting groups are the following.

Group 1	All legal moves in the proximity of the last move.
Group 2	All legal moves on the border of the game board which do not belong to group 1.
Group 3	All legal moves which do not belong to either group 1 or group 2.

The experiment was conducted on a Quad-Opteron server with 3.4 GHz Opteron processors and 32 GB of memory running a well-known Linux distribution. Both algorithms are implemented in C++.

4.2 Results

Playing the 1,000 games required a total playing time of ca. 30 hours. Of these, about two thirds were required by *STD* and the remainder by *AL*. Of 1,000 games *AL* won 838 and *STD* won 162.

A qualitative analysis of several dozen sample games shows that *AL* plays more consistently than *STD*. The program seems to beat *STD* because of its tactical superiority. This suggests that *AL* takes advantage of focusing samples on local positions more often than *STD*. In contrast, *AL* seems to shift the focus and play non-local moves with a better timing than *STD*. *AL* never plays border moves and does not seem to invest much effort on testing such moves during the first 50 moves, whereas *STD* occasionally plays border moves.

The result of the experiment clearly shows that Alternating-Layer UCT outperforms palin UCT. We may conclude that group nodes can serve to integrate domain knowledge in the MCTS framework successfully. Moreover, it may be inferred that the UCT framework is sufficiently flexible to choose the right group nodes, and that providing group nodes significantly improves the ability of the program to focus on promising branches more quickly.

The computational overhead for grouping nodes is outweighed by the benefits of narrowing down the search space in the experiment. While this is true for the three computationally cheap feature groups tested in the experiment, it remains to be seen how well this approach scales to a larger number of groups.

5 Conclusion and Future Research

In this article we introduced the concept of *group nodes* for MCTS. Alternating-Layer UCT was proposed as a technique for adopting group nodes for MCTS. A self-play experiment showed that Alternating-Layer UCT outperformed standard UCT. Based on the outcome of the experiment we may tentatively conclude that the proposed approach can incorporate domain-specific knowledge in MCTS successfully.

Future work will address the following six items. First, in order to examine whether the results found in this work generalize to deeper search, the programs will be tested with a more generous time setting. Second, the number of groups will be increased using more refined features, e.g., by using a move predictor. Third, the weighting of the probabilities assigned to group nodes will be examined more closely. Fourth, other algorithms for including group nodes in the MCTS framework could be devised. Whereas the Alternating-Level UCT straightforwardly adds group nodes after every node expansion, it might prove more useful to expand group nodes only for certain move nodes. This might reduce the computational cost required for grouping nodes. Similarly, group nodes could be allowed to have group nodes as their child nodes. Fifth, the grouping techniques will be compared to other means of incorporating domain knowledge in UCT. Sixth, the new technique will be implemented in a tournament program, e.g., MANGO.

Acknowledgements

We are indepted to Ulaş Türkmen and Guillaume Chaslot for their valuable feedback. This work is financed by the Dutch Organization for Scientific Research, NWO, as part of the project GO FOR GO, grant number 612.066.409.

References

1. Bouzy, B.: Move Pruning Techniques for Monte-Carlo Go. In van den Herik, H.J., Hsu, S.C., sheng Hsu, T., Donkers, J.H., eds.: Advances in Computer Games (ACG 11). Volume 4250 of LNCS., Springer-Verlag, Berlin (2006) 104–119

2. Bouzy, B., Helmstetter, B.: Monte Carlo Developments. In van den Herik, H.J., Iida, H., Heinz, E.A., eds.: Proceedings of the Advances in Computer Games Conference (ACG 10), Kluwer Academic (2003) 159–174
3. Brügmann, B.: Monte Carlo Go. White paper (1993) http://www.ideanest.com/vegos/MonteCarloGo.pdf.
4. Chaslot, G., Saito, J.T., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Monte-Carlo Strategies for Computer Go. In Schobbens, P.Y., Vanhoof, W., Schwanen, G., eds.: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium. (2006) 83–91
5. Chaslot, G., Winands, M.H.M., Bouzy, B., Uiterwijk, J.W.H.M., van den Herik, H.J.: Progressive Strategies for Monte-Carlo Tree Search. (2007) Submitted to JCIS 2007.
6. Ciancarini, P., Favini, G.P.: A Program to Play Kriegspiel. ICGA Journal **30**(1) (2007) 3–24
7. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Ciancarini, P., van den Herik, H.J., Donkers, J.H., eds.: Proceedings of the Fifth Computers and Games Conference. LNCS, Springer-Verlag, Berlin (2007) 12 pages, in print.
8. Kocsis, L., Szepesvári, C.: Bandit Based Monte-Carlo Planning. In Fürnkranz, J., Scheffer, T., Spiliopoulou, M., eds.: Proceedings of the EMCL 2006. Volume 4212 of LNLCS., Springer-Verlag, Berlin (2006) 282–293
9. Sakuta, M., Iida, H.: Solving Kriegspiel-like Problems: Exploiting a Transposition Table. ICGA Journal **23**(4) (2000) 218–229
10. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-Tree Search Algorithm Based on Realization Probability. ICGA Journal **25**(3) (2002) 145–152

Other Abstract Games

An Efficient Approach to Solve Mastermind Optimally

Li-Te Huang[1], Shan-Tai Chen[2], Shih-Chieh Huang[1], and Shun-Shii Lin[1,☆]

[1] Graduate Institute of Computer Science and Information Engineering, National Taiwan Normal University, No. 88, Sec. 4, Ting-Chow Rd., Taipei, Taiwan, R.O.C.
[2] Department of Computer Science, Chung Cheng Institute of Technology, National Defense University, Tao-Yuan, Taiwan, R.O.C.
☆Corresponding author: linss@csie.ntnu.edu.tw

Abstract. The Mastermind game is well-known around the world. In recent decades, several approaches have been adopted for solving Mastermind in the worst case, and an optimal strategy has finally been proposed by Koyoma and Lai in 1993 by using an exhaustive search for finding the optimal strategy in the expected case. In this paper, a more efficient backtracking algorithm with branch-and-bound pruning (BABBP) for Mastermind in the expected case is introduced, and an alternative optimal strategy is obtained eventually. Furthermore, the novel approach may be presumably applied to other games with some modifications in order to speed up the search.

1 Introduction

Mastermind, a kind of deductive games, is well-known around the world since its appearance in 1972. It has attracted considerable interests in recent decades. Two players are involved in the game. One is a codemaker and the other is a codebreaker. A secret code, which consists of 4 digits with 6 possible symbols, e.g., 1, 2, …, 6, is chosen by the codemaker and repeated symbols are allowed. The mission of the codebreaker is to obtain the code by guessing iteratively without any information before he/she starts. Based on the responses, the codebreaker has to minimize the number of guesses needed.

In more precise notation, suppose that the codemaker chooses the secret code, $s = s_1s_2s_3s_4$ and the codebreaker takes a guess, called $g = g_1g_2g_3g_4$. Then, the codebreaker gets the hint [B, W]. The variables, B and W, are calculated in the following definitions.

- B means the number of "direct hits", i.e., $B = \left|\{i : s_i = g_i\}\right|$.
- W is the number of "indirect hits", i.e., $W = \sum_{j=1}^{6} \min(p_j, q_j) - B$, where $p_j = \left|\{i : s_i = j\}\right|$ and $q_j = \left|\{i : g_i = j\}\right|$.

For example, the codemaker gives the hint, [2, 1], if the secret code s is 1123 while the guess g made by the codebreaker is 3124.

A strategy for minimizing the number of guesses was first proposed by Knuth [4]. His approach achieves the optimal result in the worst case, where the maximal number of guesses needed is 5. Meanwhile, its number of guesses in the expected case is 4.478. Much research on finding the optimal strategy in the expected case arose from then on. Irving [3] and Neuwirth [8] made improvements on the bounds, which are 4.369 and 4.364, respectively. Finally Koyoma and Lai [6] introduced an optimal strategy in the expected case for it while the expected number of guesses is about 4.34. A comprehensive introduction to Mastermind and a new strategy were demonstrated by Kooi [5]. Chen et al. [1] proved the exact bound of "Bull and Cow" in the worst case, which is also a kind of deductive games and is popular in England and Asia.

The organization of this paper is as follows. In section 2, some definitions are described. Section 3 investigates our backtracking algorithm with branch-and-bound pruning for Mastermind. In section 4, some parameters and experimental results are presented. Section 5 exhibits our concluding remarks and some critical issues in the future research.

2 Definitions and Notations

There are two issues for optimizing deductive-game problems. One is to minimize the guesses made by the codebreaker in the worst case, and the other is to minimize that in the expected case. ***An optimal strategy in the worst case*** is a strategy which minimizes the maximum number of guesses needed by the codebreaker for any secret code chosen by the codemaker. ***An optimal strategy in the expected case*** is a strategy which minimizes the expected number of guesses required with considerations of all possible codes. Note that a uniform distribution over all the codes is assumed.

An alternative aspect of viewing the optimization for deductive games as a game-tree search is adopted in this paper. We have the following definitions.

Definition 1. During the gaming process, the set of remaining candidates, which means all possible codes until now, is referred to a *state*.

Definition 2. A *final state* (leaf) is a state whose size is equal to 0. In other words, no codes remain in the final state and the game is over.

Definition 3. The *height* of the game tree of a deductive game is exactly the maximal number of guesses required by the codebreaker, i.e., the length of its longest path from the root to a leaf.

Definition 4. The number of guesses needed in the expected case is L / n, where L is the *external path length* of the game tree and n is the number of all possible codes.

Here is an example to illustrate the above definitions. Let's consider a simple guessing game with two players, which can be viewed as a simplified version of Mastermind. The codemaker chooses a secret code from the set $\{1, 2, 3, 4\}$. A hint, which is one of <, =, and >, is received by the codebreaker in each ply. Accordingly, the codebreaker has to guess the code by taking those hints into account. Figure 1 is a game tree for the game.

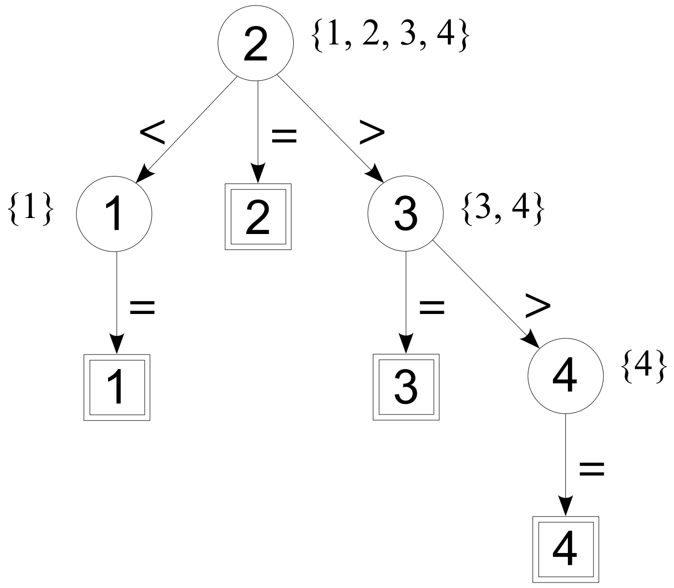


Fig. 1. An illustrative example.

Some observations are revealed from Figure 1. First of all, the state of the root is {1, 2, 3, 4} because there are four possible codes at the beginning. And it is easy to notice that the 4 leaves are final states. The height of the game tree is 3 because at most 3 guesses are required by the codebreaker. Furthermore, Figure 1 also shows that the external path length is $1\times1 + 2\times2 + 3\times1 = 8$.

3 A Backtracking Algorithm with Branch-and-bound Pruning

A large number of real-world problems can be modeled as optimization problems or games. Search algorithm is therefore a general approach for them. Unfortunately, most of these problems are NP-hard or PSPACE. In other words, it has to take exponential time to search for an optimal solution. Thus, there are plenty of pruning techniques published in the literature such as A^* search [9], branch-and-bound pruning [7], etc.

Previous pruning approaches are appropriate for optimization problems since their goal is to find a best solution in the search space. So, the search ends when it is found. A complete search is theoretically required to our problem because of the considerations of the optimal strategy in the expected case. Hence, traditional pruning approaches may not easily be applied to our problem directly.

A novel pruning technique based on the admissible heuristic in the A^* search is proposed to solve our problem. In Section 3.1, the framework of our backtracking algorithm with branch-and-bound pruning (abbreviated to BABBP) is introduced. Section 3.2 illustrates the detailed operations of our scheme.

3.1 The Framework of Our Scheme

The idea of our scheme is similar to the admissible heuristic in the A^* search. The A^* search is a tree (graph) search algorithm which finds a best path from a given initial state to a given goal with the lowest cost. The algorithm will terminate if a best solution is found. However, a complete search is conceptually required for our problem. Hence, BABBP will search the whole game tree and prune the unnecessary states by using an admissible heuristic.

Suppose that h' is the cost from the root to the current state and h^* is the cost from the current state to the final state. Then, h^* is called *admissible* if it never overestimates the cost to reach the final state. In other words, the actual cost is less than or equal to $h' + h^*$. It can also be viewed as a theoretical lower bound for the problem we deal with.

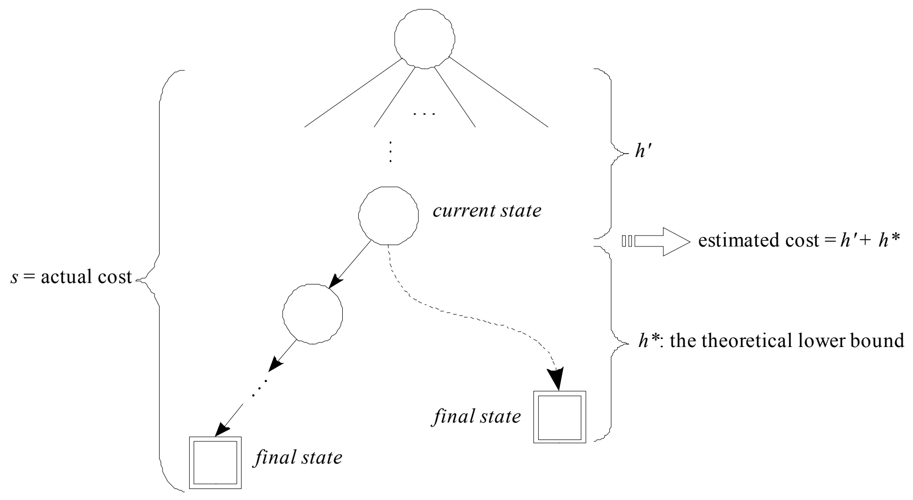


Fig. 2. The scenario of branch-and-bound pruning.

Our scheme traverses the game tree in depth-first fashion until a final state is reached. It then gets an actual cost s which is initially assigned to be the current-best solution. It soon backtracks to its parent and picks one of the other child states and uses an admissible heuristic to estimate the cost h^*. The search continues if s is larger than $h' + h^*$. Otherwise, a cut happens because s is less than or equal to $h' + h^*$. In other words, there is no need to visit the child state and its sub-tree and the correctness of the algorithm is still maintained. Similar manner continues until the search to the whole game tree is finished.

Figure 2 shows a scenario. The current state is a state which we consider currently. The admissible heuristic will estimate its cost h^* and we compare $h' + h^*$ with the actual cost s to decide whether it is able to be cut or not.

A rough sketch of the whole algorithm is exhibited in Figure 3. It is especially important to notice that BABBP always maintains a current-best solution s during the search. Hence, BABBP goes through the downward direction at first until a final state is reached. It therefore gets a current-best solution (s is updated). Then, BABBP backtracks and starts to estimate h^* in each of the following states. Unnecessary states will never be visited. Note that it updates s constantly when final states are encountered. So, BABBP will finally obtain an optimal solution when the whole game tree has been traversed completely.

```
    BABBP (state v) {
01  if (a final state is reached) then              // Final state indicates the leaf of the
      return the current-best solution s;              game tree.
02  visit and expand v;
03  for (each child u of v) {                       // Each u is a child state of v.
04    h* = ESTIMATE( u);                            // ESTIMATE is an admissible
                                                       heuristic of predicting the cost
                                                       from u to a final state.
05    if (h' + h* < s) then                         // h' is the actual cost from the start
                                                       state to u.
06      BABBP( u);                                  // Search recursively from the state u.
07    else
08      Cut the child u;                            // A cut happens if h' + h* ≥ s.
09  }
10 }
```

Fig. 3. The sketch of the backtracking algorithm with branch-and-bound pruning.

3.2 BABBP for Mastermind in the Expected Case

In this section, we will deal with Mastermind in the expected case. The search space for Mastermind is first analyzed and the admissible heuristic is designed carefully. At last, an optimal strategy is found as a result of applying BABBP to this problem.

We now investigate how large search space is for our problem. For a deductive game, the branching factor for each ply depends on the number of possible hints r and possible symbols c of the game. Accordingly, the branching factor $b = r \times c$, and hence, the search space equals to $(r \times c)^h$, where h is the height of the game tree. For instance, the search space for Mastermind is $(14 \times 1296)^h$ because the codebreaker has $6^4 = 1296$ possible guesses in each ply while the codemaker has 14 possible hints (also called classes).

It takes much time to find an optimal strategy by searching the game tree completely. A pruning technique adopted by BABBP is used to save a lot of time instead of making an exhaustive search. Figure 4 shows the game tree of Mastermind by applying BABBP. The circles in the figure mean the states which are the sets of remaining candidates while the diamonds are the possible guesses the codebreaker can choose (1296 possible guesses in each ply). In the game tree, the 14 ways produced by the codemaker's hints should be traversed completely and the 1296 ways expanded by the codebreaker may be pruned by the admissible heuristic since we are aimed to find an optimal strategy for the codebreaker. Let's consider a situation exhibited in Figure 4. The traversal to the sub-trees of g_1 (in bold style) is just finished and g_2 is now taken into account. An estimated value h^* is obtained with the use of ESTIMATE function mentioned in the previous section. The sub-trees below g_2 do not have to be expanded if the result of expanding g_1 is better than h^*. This is the key idea of BABBP and the search can thus be completed in a more reasonable time. Note that the correctness of BABBP is reserved because of the admissible heuristic.

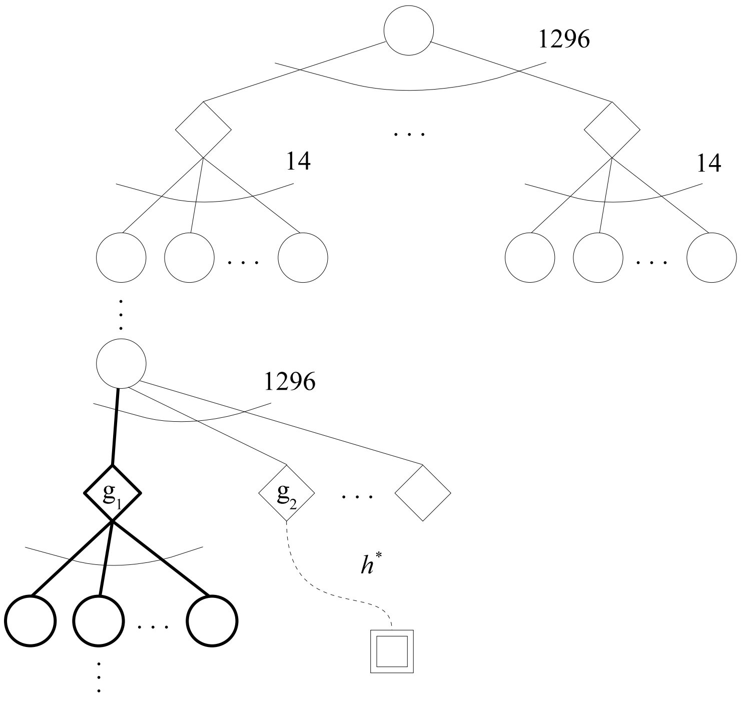


Fig. 4. The game tree of Mastermind by applying BABBP.

Now the most critical issue is how to design an admissible heuristic function to estimate the theoretical lower bound h^*. Note that minimizing the number of guesses in the expected case is the same as minimizing the external path length of the game tree. So, the concept of volumes introduced in [2] is involved to get the theoretical maximum bounds for the 14 classes (responses). We know that different guesses in some ply result in distinct distributions of the remaining candidates in 14 classes. The volume of a class (response) [x, y] is defined as the maximum value of the numbers of the remaining candidates when the codebreaker makes all the possible guesses in one ply and the codemaker responses with [x, y]. In the beginning, at the root of Figure 4, there are totally 1296 remaining candidates (guesses). While the codebreaker makes the first guess, there are 5 nonequivalent guesses in 1296 possible codes, i.e., “1111”, “1112”, “1122”, “1123”, and “1234”. If the codebreaker guesses “1111” and the codemaker gives the hint [1, 0], then we can derive that there are 500 possible secret codes. Similarly, if the code-breaker guesses “1112”, “1122”, “1123”, or “1234”, and the code-maker gives the hint [1, 0], then we can derive that there are 317, 256, 182, and 108 possible secret codes, respectively. So, the volume of the class [1, 0] is set to be 500, the maximum value of these numbers: 500, 317, 256, 182, and 108. With the use of Get_volume function [2] based on the above idea, the volumes of the 14 classes (responses) are obtained as in Table 1.

Table 1. The volumes of 14 classes calculated by Get_volume function.

Class	[4, 0]	[2, 2]	[1, 3]	[0, 4]	[3, 0]	[2, 1]	[1, 2]	[0, 3]	[2, 0]	[1, 1]	[0, 1]	[0, 2]	[1, 0]	[0, 0]
volume	1	6	8	9	20	48	132	136	150	252	308	312	500	625

We therefore employ the same principle of the extended pigeonhole principle presented by [1] to estimate the lower bounds of the guesses needed among 14 classes which mean 14 responses. However, there are major differences between the problem in [1] and this problem we consider now. Only the worst case among the 14 classes is considered for the codemaker in [1]. The so-called “worst case” denotes the class which will result in the most guesses needed by the codebreaker. But the distribution of each class should be taken into account for our problem. Moreover, the heuristic function here has to calculate the “theoretical optimal” strategy in the expected case for the codebreaker. In other words, it will assume that there exists an optimal strategy such that all of the remaining candidates in each class may be divided evenly in the following guesses. The actual expected number of guesses is thus more than or equal to the value of estimations. A simple example to illustrate the calculation of the admissible heuristic function is shown in Figure 5.

Giving a state with a size of 17, as shown in Figure 5, we imagine that the theoretical optimal strategy will divide the 17 candidates into 14 classes evenly without exceeding the corresponding volumes. The number in the lower half of the circle is the volume of each class and the number in the upper half is the number of candidates in it. Since there are 1 leaf at level 1, 13 leaves at level 2, and 3 leaves at level 3, it is obvious that the external path length of the tree is 1×1 + 2×13 + 3×3 = 36 in the ideal situation.

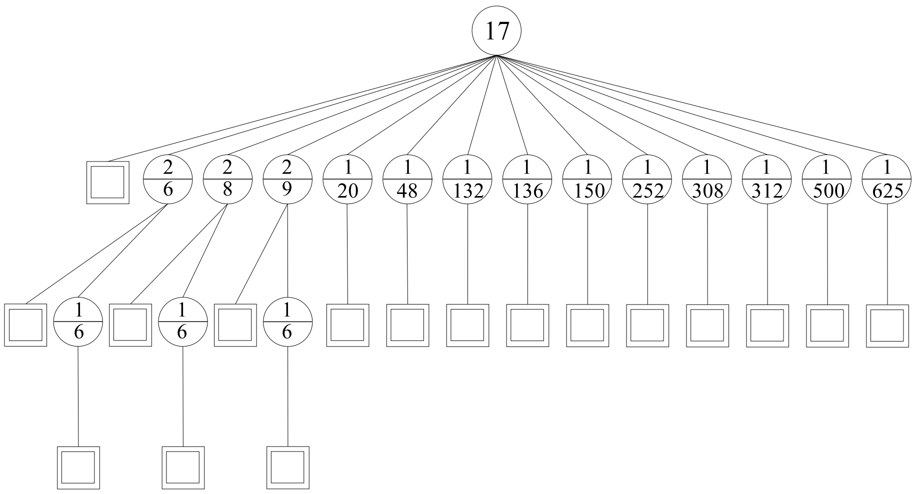


Fig. 5. An illustrative example for the calculation of the admissible heuristic function.

4 Experimental Results

In order to analyze the performance of the proposed BABBP, we demonstrate the results of the original Mastermind (4×6 Mastermind) and another version of Mastermind which is called 3×5 Mastermind. 3×5 Mastermind has smaller search space in the case of 3 digits with 5 possible symbols. That is to say that it has $5^3 = 125$ possible secret codes totally. Note that the equivalent properties proposed in [8] are able to reduce the search space. For example, 1111 is equivalent to 2222 in the first guess because the numbers, 1 and 2, are both not used before. With the considerations of the properties, there are five unequivalent guesses in the first guess, which are 1111, 1112, 1122, 1123, and 1234. The branching factor in the first ply changes from 14×1296 to 14×5 eventually. This technique has also been implemented in our programs in order to speed up the search.

Table 2. The experimental results of two versions of Mastermind.

	3×5 Mastermind	4×6 Mastermind
DFS	> 10 hr.	> 10 days
BABBP	38.68 sec.	43.76 hr.
BABBP (sorted order)	11.21 sec.	9.5 hr.
External path length	451	5625

Besides the comparison between 3×5 Mastermind and 4×6 Mastermind, we also investigate the effect of the traversing order during the search. In other words, we have to decide which child state should be traversed first when several child states are encountered after the current state is visited. To deal with this issue, we estimate the lower bounds of the child states by making use of the admissible heuristic before they are expanded. We sort their lower bounds and traverse these states order-by-order in accordance with their values. All experiments were run on a dedicated PC with an AMD Opteron 252 processor. The experimental results are exhibited in Table 2.

DFS is the abbreviation of depth-first search while the term, "sorted order", means that BABBP visits the child states in nondecreasing order according to the values of lower bounds. We can see that BABBP is able to obtain the optimal strategies for the two versions and their corresponding external path length is 451 and 5625, respectively. This means that the expected number of guesses is about 4.34 (=5625/1296) for Mastermind if we apply the optimal strategy in the expected case. The results also show that BABBP with the considerations of expanding order has the most outstanding performance. Without the technique of ordering, BABBP will visit a lot of useless states. That is to say that most states will be cut if BABBP expands in the correct order. On the other hand, DFS has very poor performance doubtlessly since it is certainly an exhaustive search. Hence, the larger the search space is, the more important the pruning technique is.

5 Conclusions

Previously, an exhaustive search was applied to find the optimal strategy for Mastermind. But it may not be adopted in other larger problems or games because of its huge search time. In this paper, a more efficient backtracking algorithm with branch-and-bound pruning (BABBP) for Mastermind in the expected case is introduced, and an alternative optimal strategy is obtained eventually. The effect of expanding order during the search is significant to the performance of BABBP. How to design a more precise heuristic function is yet another critical issue. We hope that the approach may be applied to other related deductive games in the future.

Acknowledgments. The authors would like to thank anonymous referees for their valuable comments. The research is supported in part by a grant 95-2221-E-003-009, ROC.

References

1. Chen, S.T., Lin, S.S., Huang, L.T., Hsu, S.H.: Strategy optimization for deductive games. European Journal of Operational Research (2006) doi:10.1016/j.ejor.2006.08.058
2. Huang, L.T., Chen, S.T., Lin, S.S.: Exact-bound analyses and optimal strategies for Mastermind with a lie. Lecture Notes in Computer Science, 11th Advances in Computer Games, Vol. 4250. Springer-Verlag, Berlin Heidelberg (2006) 195–209
3. Irving, R.W.: Towards an optimum Mastermind strategy. Journal of Recreational Mathematics, Vol. 11. No. 2. Baywood Publishing Company, Inc. (1978-79) 81–87
4. Knuth, D.E.: The computer as Mastermind. Journal of Recreational Mathematics, Vol. 9. No. 1. Baywood Publishing Company, Inc. (1976) 1–6
5. Kooi, B.: Yet another Mastermind strategy. International Computer Games Association Journal, Vol. 28. No. 1. International Computer Games Association (2005) 13–20

6. Koyama, K., Lai, T.W.: An optimal Mastermind strategy. Journal of Recreational Mathematics, Vol. 25. No. 4. Baywood Publishing Company, Inc. (1993) 251–256
7. Neapolitan, R., Naimipour, K.: Foundations of Algorithms Using C++ Pseudocode. 3rd edn. Jones and Bartlett Publishers (2004)
8. Neuwirth, E.: Some strategies for Mastermind. Mathematical Methods of Operations Research, Vol. 26. No. 1. Physica Verlag, An Imprint of Springer-Verlag GmbH (1982) 257–278
9. Russell S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edn. Prentice-Hall (2002)

A Retrograde Approximation Algorithm for Two-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, Maryland 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. A two-player, finite, probabilistic game with perfect information can be presented as a four-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. In a previous paper we have presented our success on tackling one-player Can't Stop. In this paper we prove the existence and uniqueness of the solution to two-player Can't Stop, and present a retrograde approximation algorithm to solve it by incorporating the 2-dimensional Newton's method with retrograde analysis. We give results of small versions of two-player Can't Stop.

1 Introduction

Retrograde analysis has been well developed for deterministic and two-player zero-sum games with perfect information, and successfully applied to construct endgame databases of checkers [7, 9], chess [10, 11], and Chinese chess [1, 2, 13]. It also played a crucial role in solving Nine Men's Morris [3] and Kalah [6]. A recent success of parallel retrograde analysis was solving Awari [8].

On the other hand, retrograde analysis for probabilistic games is currently under-explored. Glenn [4] and Woodward [12] solved one-player Yahtzee. We presented our success on tackling one-player Can't Stop[1], by incorporating Newton's method into a retrograde algorithm run on a bipartite graph representing the game [5].

[1] Can't Stop was designed by Sid Sackson and marketed first by Parker Brothers and now by Face 2 Face Games. The rules can be found at `http://en.wikipedia.org/wiki/Can't_Stop` and `http://www.boardgamegeek.com/game/41`.

A two-player probabilistic game can be represented as a four-partite graph $G = (U, V, \bar{U}, \bar{V}, E)$, where $U/\bar{U}$ correspond to random events and $V/\bar{V}$ correspond to deterministic events of the first/second players, respectively. For Yahtzee, the graph representation is acyclic that simplifies algorithm design. 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 generalize our retrograde approximation algorithm for one-player Can't Stop by incorporating the 2-dimensional Newton's method into a retrograde algorithm.

The organization of this paper is as follows. Section 2 formulates the problem. Section 3 proves that two-player Can't Stop has a unique solution, and gives a retrograde algorithm to solve it. Section 4 presents the indexing scheme. Section 5 summarizes the results of the experimental tests. A conclusion is given in Section 6.

2 Problem Formulation

A two-player probabilistic game can be presented as a four-partite graph $G = (U, V, \bar{U}, \bar{V}, E)$, where $E \subseteq (U \times V) \cup (\bar{U} \times \bar{V}) \cup ((V \cup \bar{V}) \times (U \cup \bar{U}))$. Edges in $U \times V$ and $\bar{U} \times \bar{V}$ represent the random events (e.g., dice rolls), and edges in $V \times (U \cup \bar{U})$ and $\bar{V} \times (U \cup \bar{U})$ represent the deterministic events (i.e., the moves), by the first and second players, respectively. We also call vertices in $U \cup \bar{U}$ *roll positions* and vertices in $V \cup \bar{V}$ *move positions*. A terminal vertex indicates the end of a game. Without loss of generality, we assume all terminal vertices are roll positions (i.e., in $U \cup \bar{U}$).

A *partial turn* (u_1, u_2) (from roll position u_1 to roll position u_2) consists of a random event followed by a move. It is represented by a pair of edges $((u_1, v), (v, u_2))$ in G. A sequence of partial turns $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$ is called a *turn*. In Can't Stop, a turn may consist of many partial turns. In Yahtzee a turn consists of exactly one partial turn, and hence $E \subseteq (U \times V) \cup (V \times \bar{U}) \cup (\bar{U} \times \bar{V}) \cup (\bar{V} \times U)$.

We associate each position with a real number representing the expected game score that the first player achieves in optimal play, denoted by a function $f : U \cup \bar{U} \cup V \cup \bar{V} \rightarrow R$. For two-player Can't Stop, $f(u)$ indicates the probability that the first player will win the game. Since the games we consider are zero-sum, the probability that the second player wins is $1 - f(u)$. For any terminal position $z \in U \cup \bar{U}$, $f(z) = 1$ if the first player wins, and $f(z) = 0$ if the first player loses. For two-player Yahtzee, a game may end in a draw. In this case we can set $f(z) = 0.5$. However, it depends on the goal of the two players. If the goal of the first player is either to win or to draw the game, we set $f(z) = 1$ for draw positions. Assuming the zero-sum property holds, the goal of the second player is to win the game. On the other hand, if a draw means nothing different from a loss to the first player, we set $f(z) = 0$ for draw positions.

For each non-terminal roll position $u \in U \cup \bar{U}$, the weight $0 < p((u, v)) \leq 1$ indicates the probability that the game in u will change into move position v.

Therefore,

$$\sum_{\forall v \text{ with } (u,v)\in E} p((u,v)) = 1.$$

Recall that $f(u)$ is the expected score that the first player achieves. For each non-terminal roll position $u \in U \cup \bar{U}$,

$$f(u) = \sum_{\forall v \text{ with } (u,v)\in E} p((u,v))f(v). \tag{1}$$

In optimal play, the first player maximizes f whereas the second player minimizes f. Therefore, for all non-terminal move positions $v \in V \cup \bar{V}$,

$$f(v) = \begin{cases} \max\{f(u) : (v,u) \in E\} & \text{if } v \in V, \\ \min\{f(u) : (v,u) \in E\} & \text{if } v \in \bar{V}. \end{cases} \tag{2}$$

For all positions $w \in U \cup \bar{U} \cup V \cup \bar{V}$, $f(w)$ is also called the *position value* of w.

A database f satisfying both conditions (1) and (2) is called a *solution*. A game is *solved* if a solution is obtained. Unless otherwise noted, all the game graphs in this paper represent finite, zero-sum, two-player, probabilistic games with perfect information, abstracted as above.

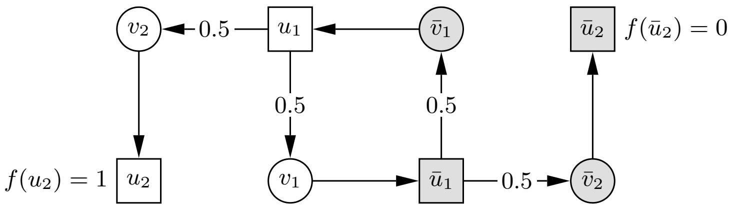


Fig. 1. An example of two-player game graph $G = (U, V, \bar{U}, \bar{V}, E)$.

We illustrate an example in Figure 1, where $u_1, u_2 \in U$, $v_1, v_2 \in V$, $\bar{u}_1, \bar{u}_2 \in \bar{U}$, and $\bar{v}_1, \bar{v}_2 \in \bar{V}$. The two terminal vertices are u_2 and $\bar{u}_2$ with $f(u_2) = 1$ and $f(\bar{u}_2) = 0$, respectively. This example simulates the last stage of a game of two-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 player is in the same situation at position $\bar{u}_1$. By (1) and (2),

$$\begin{array}{lll} f(u_1) = \frac{1}{2}f(v_1) + \frac{1}{2}f(v_2), & f(v_1) = f(\bar{u}_1), & f(v_2) = f(u_2) = 1, \\ f(\bar{u}_1) = \frac{1}{2}f(\bar{v}_1) + \frac{1}{2}f(\bar{v}_2), & f(\bar{v}_1) = f(u_1), & f(\bar{v}_2) = f(\bar{u}_2) = 0. \end{array}$$

The unique solution is $f(u_1) = f(\bar{v}_1) = \frac{2}{3}$ and $f(u_2) = f(\bar{v}_2) = \frac{1}{3}$.

The problem of solving a two-player probabilistic game is formulated as follows. Suppose we are given a game graph $G = (U, V, \bar{U}, \bar{V}, E)$ with function values of the terminal vertices well-defined. First, we investigate the existence and uniqueness of the solution. Second, we design an efficient algorithm to construct the database f, assuming a solution exists.

3 Retrograde Analysis for Two-Player Probabilistic Games

A retrograde algorithm typically consists of three phases: initialization phase, propagation phase, and the final phase. In the initialization phase, the terminal vertices are associated with their position values. In the propagation phase, the information is propagated iteratively back to its predecessors until no propagation is possible. The final phase deals with the undetermined vertices.

Subsection 3.1 gives an algorithm to construct the database of an acyclic game graph. In Subsection 3.2 we prove that two-player Can't Stop has a unique solution. In Subsection 3.3 we develop an algorithm to construct the database for a game graph with cycles.

3.1 Game Graph is Acyclic

For games with acyclic game graphs, such as two-player Yahtzee, the bottom-up propagation procedure is clear. Algorithm 1 gives the pseudocode to construct the database for an acyclic game graph. It can also be applied to constructing the perfect Backgammon bear-off databases with no piece on the bar[2].

Consider Algorithm 1. Assuming all terminal vertices are in $U \cup \bar{U}$, the set S_2 is initially 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 $v \in V$, always make the move (v, u) with the maximum $f(u)$. For $\bar{v} \in \bar{V}$, we make the move $(\bar{v}, \bar{u})$ with the minimum $f(\bar{u})$. The proof of Lemma 1 reveals that the acyclic property ensures all vertices are determined at the end of propagation phase. Therefore, a final phase is not required.

Lemma 1. *If a game graph is acyclic, its solution exists and is unique.*

Proof. 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 the solution exists and is unique. □

[2] To save constructing time and database space, we may use the optimal solution of the simplified one-player game for the approximate Backgammon bear-off databases with no piece on the bar.

Algorithm 1 Construct database f for an acyclic game graph.

```
Require: $G = (U, V, \bar{U}, \bar{V}, E)$ is acyclic.
Ensure: Program terminates with (1) and (2) satisfied.                ▷ Lemma 1
  $\forall u \in U \cup \bar{U}$, $f(u) \leftarrow 0$.                 ▷ Initialization Phase
  $\forall v \in V$, $f(v) \leftarrow -\infty$, $\forall v \in \bar{V}$, $f(v) \leftarrow \infty$.
  $S_1 \leftarrow \{$terminal positions in $U \cup \bar{U}\}$
  $S_2 \leftarrow \{$terminal positions in $V \cup \bar{V}\}$          ▷ (†)
  $\forall u \in S_1 \cup S_2$, set $f(u)$ to be its value.
  repeat                                                               ▷ Propagation Phase
      for all $u \in S_1$ and $(v, u) \in E$ do
          if $v \in V$ then
              $f(v) \leftarrow \max\{f(v), f(u)\}$
          else[$v \in \bar{V}$]
              $f(v) \leftarrow \min\{f(v), f(u)\}$
          end if
          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 $S_1 \cup S_2 = \emptyset$
```

Note that 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.

3.2 Game Graph is Cyclic

If a game graph is cyclic, a solution may not exist. Even if it exists, it may not be unique. We give a condition under which a solution exists and is unique in Lemma 2. The proof uses the Fixed Point Theorem[3]. With Lemma 2, we

[3] See, e.g., `http://mathworld.wolfram.com/FixedPointTheorem.html`.

prove that the game graph of two-player Can't Stop has a unique solution in Theorem 2.

Theorem 1 (Fixed Point Theorem). *If a continuous function $f : R \longrightarrow R$ satisfies $f(x) \in [a,b]$ for all $x \in [a,b]$, then f has a fixed point in $[a,b]$ (i.e., $f(c) = c$ for some $c \in [a,b]$).*

Lemma 2. *Given a cyclic two-player game graph $G = (U, V, \bar{U}, \bar{V}, E)$, we use G_1 and G_2 to denote the two subgraphs of G induced by $U \cup V$ and $\bar{U} \cup \bar{V}$, respectively. G has a solution with all position values in $[0,1]$ if,*

1. *The graphs G_1 and G_2 are acyclic.*
2. *There exist $w_1 \in U$ and $w_2 \in \bar{U}$ such that all edges from G_1 to G_2 end at w_2, and all edges from G_2 to G_1 end at w_1. In other words, $E \cap (V \times \bar{U}) \subseteq V \times \{w_2\}$ and $E \cap (\bar{V} \times U) \subseteq \bar{V} \times \{w_1\}$.*
3. *All the terminal position values are in $[0,1]$.*

In addition, if there is a path in G_1 from w_1 to a terminal vertex $z \in U \cup V$ with position value $f(z) = 1$, then the solution has all position values in $[0,1]$ and is unique.

Proof. Let $\hat{G}_1 = (U \cup \{w_2\}, V, E_1)$ and $\hat{G}_2 = (\bar{U} \cup \{w_1\}, \bar{V}, E_2)$ be the induced bipartite subgraphs of G. By condition 1, $\hat{G}_1$ and $\hat{G}_2$ are acyclic. Consider $\hat{G}_1$. All the terminal vertices in $\hat{G}_1$ other than w_2 are also terminal in G. If we know the position value of w_2, then by Lemma 1 the solution to $\hat{G}_1$ can be uniquely determined. Let y be the estimated position value of w_2. We can construct a database for $\hat{G}_1$ by Algorithm 1. Denote by $\hat{f}_1(y, w)$ the position value of $w \in U \cup V$ that depends on y. Likewise, given x as the estimated position value of w_1, we denote by $\hat{f}_2(x, \bar{w})$ the position value of $\bar{w} \in \bar{U} \cup \bar{V}$. The values of $\hat{f}_1(y, w)$ for $w \in U \cup V$ and $\hat{f}_2(x, \bar{w})$ for $\bar{w} \in \bar{U} \cup \bar{V}$ constitute a solution to G, if and only if $\hat{f}_1(y, w_1) = 0$ and $\hat{f}_2(x, w_2) = 0$. The main theme of this proof is to discuss the existence and uniqueness of x and y satisfying $\hat{f}_1(y, w_1) = x$ and $\hat{f}_2(x, w_2) = y$, or equivalently $\hat{f}_2(\hat{f}_1(y, w_1), w_2) = y$.

Condition 3 states that all terminal position values are in $[0,1]$. Iteratively applying (1) and (2), $\hat{f}_2(\hat{f}_1(0, w_1), w_2) \geq 0$ and $\hat{f}_2(\hat{f}_1(1, w_1), w_2) \leq 1$. By Theorem 1, there exists $y^* \in [0,1]$ such that $\hat{f}_2(\hat{f}_1(y^*, w_1), w_2) = y^*$. Iteratively applying (1) and (2) again, the position values of $w \in U \cup V$, $\hat{f}_1(y^*, w)$, are all in $[0,1]$. Likewise, the position values of $\bar{w} \in \bar{U} \cup \bar{V}$, $\hat{f}_2(x^*, \bar{w})$, are also all in $[0,1]$, where $x^* = \hat{f}_1(y^*, w_1)$.

Now we investigate the uniqueness of the solution. Consider $\hat{G}_1$, whose solution can be obtained by propagation that depends on y, the position value of w_2. For convenience of discussion, we propagate in terms of y (i.e., treat y as a variable during the propagation), even though we know the value of y. For example, assuming $y = \frac{1}{2}$, we write $\max\{\frac{2}{3}y, \frac{1}{4}y + \frac{1}{6}\} = \frac{2}{3}y$ instead of $\frac{1}{3}$. Iteratively applying (1) and (2), all propagated values of $u \in U \cup V$ are in the form $ay+b$, which represents the local function values of $\hat{f}_1(y, u)$. By condition 3,

$0 \leq a+b \leq 1$ with a, b nonnegative. We are particularly concerned with $\hat{f}_1(y, w_1)$. Analysis above shows that $\hat{f}_1(y, w_1)$ in value is piecewise linear, continuous and non-decreasing with the slope of each line segment in $[0, 1]$, and so is $\hat{f}_2(x, w_2)$ by a similar discussion. These properties are inherited by the composite function $\hat{f}_2(\hat{f}_1(y, w_1), w_2)$. The additional condition, the existence of a path in G_1 from w_1 to a terminal position with position value 1 further ensures each line segment $ay + b$ of $\hat{f}_1(y, w_1)$ has the slope $a < 1$. Hence the slope of each line segment of $\hat{f}_2(\hat{f}_1(y, w_1), w_2)$ is also less than 1. This guarantees the uniqueness of the solution in $[0, 1]$ to $\hat{f}_2(\hat{f}_1(y, w_1), w_2) = y$. □

Consider the strongly connected components of the game graph of two-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 markers for each player. The two roll positions with no marker are the *anchors* of the component. In one of them it is the turn of the first player, whereas in the other it is the second player to move. When left without a legal move, the game goes back to one of the two 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 1 (if the first player wins) or 0 (if the second player wins).

Theorem 2. *The game graph of two-player Can't Stop has a unique solution, where all the position values are in* $[0, 1]$.

Proof. The proof is by finite induction. We split the graph into strongly connected components, and consider the components in bottom-up order.

Given a non-terminal component with the anchors in its supporting components having position values uniquely determined in $[0, 1]$, we consider the subgraph induced by the component and the anchors in its supporting components. This subgraph satisfies all conditions in Lemma 2, where the terminal positions are the anchors in the supporting components. Therefore, it has a unique solution with all position values in $[0, 1]$. By induction, the solution to the game graph of two-player Can't Stop exists and is unique. □

3.3 Retrograde Approximation Algorithms

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 $G = (U, V, \bar{U}, \bar{V}, E)$, we prune some edges so the resulting $\hat{G} = (U, V, \bar{U}, \bar{V}, \hat{E})$ 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.

An example is illustrated by solving $G = (U, V, U, \bar{V}, E)$ in Figure 1. We remove $(v_1, \bar{u}_1)$ to obtain the acyclic graph $\hat{G}$, and initialize the newly terminal

Algorithm 2 A naive algorithm to solve a cyclic game graph.

```
Ensure: If $\hat{f}$ converges, it converges to a solution to $G = (U, V, \bar{U}, \bar{V}, E)$.
  Obtain an acyclic graph $\hat{G} = (U, V, \bar{U}, \bar{V}, \hat{E})$, where $\hat{E} \subset E$.   ▷ 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 $U \cup \bar{U}\}$
  $S_2 \leftarrow \{$terminal positions of $\hat{G}$ in $V \cup \bar{V}\}$
  repeat   ▷ Approximation Phase
    for all $u \in S_1$ and $(v, u) \in E$ do
      $\hat{f}(v) \leftarrow \begin{cases} \max\{\hat{f}(w) : (v, w) \in E\} & \text{if } v \in V, \\ \min\{\hat{f}(w) : (v, w) \in E\} & \text{if } v \in \bar{V}. \end{cases}$   ▷ (*)
      $S_2 \leftarrow S_2 \cup \{v\}$
    end for
    $S_1 \leftarrow \emptyset$
    for all $v \in S_2$ and $(u, v) \in E$ do
      $\hat{f}(u) \leftarrow g(u) + \sum_{\forall w \text{ with } (u,w) \in E} p((u, w))\hat{f}(w)$   ▷ (**)
      $S_1 \leftarrow S_1 \cup \{u\}$
    end for
    $S_2 \leftarrow \emptyset$
  until $\hat{f}$ converges.
```

vertex u_1 with position value 0. The solution for $\hat{G}$ has $\hat{f}(u_1) = \frac{1}{2}$. The update is repeated with $\hat{f}(u_1) = \frac{5}{8}, \frac{21}{32}, \ldots, \frac{1}{2} + \frac{1}{6}\frac{4^n - 1}{4^n}, \ldots$, which converges to $\frac{2}{3}$. Hence $\hat{f}$ converges to the solution to G. Let e_n be the difference between $\hat{f}(u_1)$ at the nth step and the converged value; then $\frac{e_{n+1}}{e_n} = \frac{1}{4}$. Hence it converges linearly.

Consider Algorithm 2. For two-player Can't Stop, it is natural to prune the outgoing edges of the anchors and obtain the acyclic $\hat{G}$. In (†), we assign an estimated value to each vertex terminal in $\hat{G}$ but not terminal in G (i.e., the newly terminal positions). For efficiency, we do not have to recompute the whole (*) and (**). Updating with the recent changes of the children is sufficient.

If the conditions in Lemma 2 are satisfied (e.g., a strongly connected component of two-player Can't Stop), $\hat{G}$ can be obtained by pruning the outgoing edges of the two anchors w_1 and w_2.

The proof of Lemma 2 reveals that if we solve $\hat{f}_1(y, w_1) = x$ and $\hat{f}_2(x, w_2) = y$ using the 2-dimensional Newton's method[4], then quadratic convergence can be expected. In other words, if we use e_n to denote the difference between the estimation and the solution at the nth step, $\frac{e_{n+1}}{e_n^2} \approx c$ for some constant c when the estimate is close enough to the solution[5]. An example is illustrated with the game graph in Figure 1 as follows. We treat u_1 as w_1 and $\bar{u}_1$ as w_2 in Lemma 2, and let x and y be the initial estimate of the position values of u_1 and $\bar{u}_1$,

[4] See, e.g., `http://www.math.gatech.edu/čarlen/2507/notes/NewtonMethod.html`.

[5] In our case $\hat{f}(x, w)$ is piecewise linear. Hence Newton's method can reach the solution in a finite number of steps. In practice, however, rounding errors may create minor inaccuracy.

respectively. Then $\hat{f}_1(y, u_1) = \frac{1}{2}y + \frac{1}{2}$, $\hat{f}_2(x, \bar{u}_1) = \frac{1}{2}x$. Solving $\frac{1}{2}y + \frac{1}{2} = x$ and $\frac{1}{2}x = y$, we obtain $x = \frac{2}{3}$ and $y = \frac{1}{3}$, which are the exact position values of u_1 and $\bar{u}_1$, respectively. In this small example we obtain the solution by one iteration. In practice, multiple iterations are expected to reach the solution. The pseudocode is given in Algorithm 3.

Algorithm 3 An efficient algorithm to solve a cyclic game graph.

```
Require: $G = (U, V, \bar{U}, \bar{V}, E)$ satisfies the conditions in Lemma 2.
Ensure: $\hat{f}_1$ and $\hat{f}_2$ converge to a solution to $G$ in the rate of Newton's method.
  {Estimation Phase:}
  Denote the induced subgraphs $\hat{G}_1 = (U \cup \{w_2\}, V, E_1)$ and $\hat{G}_2 = (\bar{U} \cup \{w_1\}, \bar{V}, E_2)$.
  Assuming the conditions in Lemma 2 hold, $\hat{G}_1$ and $\hat{G}_2$ are acyclic and $E_1 \cup E_2 = E$.
  Estimate the position values of anchors $w_1 \in U$ and $w_2 \in \bar{U}$, denoted by $x$ and $y$.
  {Approximation Phase:}
  repeat
      Solve $\hat{G}_1$ in terms of the current estimate $y$ for $w_2$ by Algorithm 1.   ▷ (*)
      Denote the linear segment of $\hat{f}_1(y, w_1)$ by $a_1 y + b_1$.
      Solve $\hat{G}_2$ in terms of the current estimate $x$ for $w_1$ by Algorithm 1.   ▷ (**)
      Denote the linear segment of $\hat{f}_2(x, w_2)$ by $a_2 x + b_2$.
      Solve $x = a_1 y + b_1$ and $y = a_2 x + b_2$ for the next estimates $x$ and $y$.
  until the values of $x$ and $y$ cannot be longer unchanged.
```

Consider Algorithm 3. In the estimation phase, the better the initial estimated position values of w_1 and w_2 (denoted by x and y respectively), the fewer steps are needed to reach the solution. In the approximation phase, the graphs $\hat{G}_1$ and $\hat{G}_2$ are disjoint except w_1 and w_2, and the propagations in (*) and (**) in each iteration are independent of each other. Therefore, Algorithm 3 is natively parallel on two processors, by separating the computations (*) and (**).

A more general model is that a game graph G has two anchors w_1, w_2 (i.e., removing the outgoing edges of w_1 and w_2 results in an acyclic graph), but the precondition in Lemma 2 does not hold. In this model the incorporation of 2-dimensional Newton's method is still possible as follows. Let x and y be the current estimated position values of w_1 and w_2 at each iteration, respectively. The propagated values of w_1 and w_2 in terms of x and y (e.g., if $x = \frac{1}{2}$ and $y = \frac{3}{4}$, we write $\min\{\frac{1}{2}x + \frac{1}{2}y, \frac{2}{3}x + \frac{1}{3}y\} = \frac{2}{3}x + \frac{1}{3}y$ instead of $\frac{7}{12}$) are denoted by $\hat{f}(x, y, w_1)$ and $\hat{f}(x, y, w_2)$. We solve the linear system of $\hat{f}(x, y, w_1) = x$ and $\hat{f}(x, y, w_2) = y$ for x and y as the position values of w_1 and w_2 in the next iteration. Three observations are worth noting. First, since the precondition in Lemma 2 does not hold, existence and uniqueness of the solution requires further investigation. Second, the algorithm is not natively parallel on two processors as stated above. Third, this generalization relies on the property of two anchors, not two players. It also applies to a one-player probabilistic game graph with two anchors.

4 Indexing Scheme

We use two different indexing schemes for Can't Stop positions, one scheme for anchors and another for non-anchors. The indexing scheme for non-anchor positions is designed so that, given an index we can quickly compute the positions of all of the markers, and vice versa. It is a mixed radix system in which there is a digit for each player and column representing the position of that player's marker in the column; this scheme is similar to that used for one-player Can't Stop [5]. A different scheme is used for anchors so that we can store the position value database in a compact form.

In the variant used in our experiments, an anchor $(x_2, ..., x_{12}, y_2, ..., y_{12}, t)$ is illegal if $x_i = y_i \neq 0$ for some i (players' markers cannot occupy the same location with a column). With this restriction many indices map to illegal anchors. Furthermore, once a column is closed, the locations of the markers in that column are irrelevant; only which player won matters. For example, if $y_2 = 3$ then we can set $x_2 = 0$ and the resulting position represents the position where $x_2 \in \{1, 2\}$ as well. If the position values are stored in an array indexed using the mixed radix system as for non-anchors, then the array would be sparse: for the official game about 98% 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.

Write each position as $((x_2, y_2), ..., (x_{12}, y_{12}), t)$. Associate with each pair (x_i, y_i) an index z_i corresponding to its position on a list of the legal pairs of locations in column i (that is, on a list of ordered pairs (x, y) such that $x \neq y$). The z_i and t are then used as digits in a mixed radix system to obtain the index

$$t + \sum_{c=2}^{12} z_c \cdot 2 \prod_{d=2}^{c-1} (3 + l_d(l_d - 1))$$

where l_d is the length of column d and the term in the product is the number of legal, distinct pairs of locations in column d. The list of ordered pairs 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 multiple columns are closed, what is important is which columns have been closed and the total number of columns won by each player, but not which player has won which columns. Before executing Algorithm 1 on a component, we check whether an equivalent component has already been solved. We deal with symmetric positions in the same way.

5 Experiments

As proof of concept, we have solved simplified versions of two-player Can't Stop. The simplified games use dice with fewer than six sides and may have shorter columns than the official version. Let (n, k) Can't Stop denote the two-player game played with n-sided dice and columns of length $k, k+2, ..., k+2(n-1), ..., k$.

We have implemented Algorithm 3 in Java and solved $(3, k)$ Can't Stop for $k = 1, 2, 3$. We used an initial estimate of $(\frac{1}{2}, \frac{1}{2})$ for the position values of the anchors within a component. Table 1 shows, for three versions of the game, the size of the game graph, the time it took the algorithm to run, and the probability that the first player wins assuming that each player plays optimally. The listed totals for components and positions within those components excludes the components not examined because of equivalence.

Table 1. Results of solving simple versions of Can't Stop.

(n, k)	Components	Total positions	Time	P(P1 wins)
$(3, 1)$	6,324	634,756	4m33s	0.760
$(3, 2)$	83,964	20,834,282	3h45m	0.711
$(3, 3)$	930,756	453,310,692	3d13h	0.689

Note that the time to solve the game grows faster than the number of positions. This is because the running time is also dependent on the number of iterations per component, which is related to the quality of the initial estimate and the complexity of the game. Table 2 gives the average number of iterations versus the position value of the component, given as (x, y) where x (y) is the probability that the first player wins given that the game has entered the component and it is the first (second) player's turn. Note that the table is upper triangular because there is never an advantage in losing one's turn and symmetric because of symmetric positions within the game. Perhaps surprisingly, the components that require the most iterations are not those where the solution is farthest from the initial estimate of $(\frac{1}{2}, \frac{1}{2})$. We conjecture that this is because positions where there is a large penalty for losing one's turn require less strategy (the decision will usually be be to keep rolling) and therefore $\hat{f}$ is less complicated (has fewer pieces) and so Newton's method converges faster.

6 Conclusion

We used a four-partite graph to abstract a two-player probabilistic game. Given a position, its position value indicates the winning rate of the first player in optimal play. We investigated the game of two-player Can't Stop, and proved that its optimal solution exists and is unique. To obtain the optimal solution,

Table 2. Iterations required vs. position values for (3, 3) Can't Stop

		x				
		0.0-0.2	0.2-0.4	0.4-0.6	0.6-0.8	0.8-1.0
	0.0-0.2	3.00	3.25	3.50	3.37	2.87
	0.2-0.4	-	3.27	4.12	3.83	3.37
y	0.4-0.6	-	-	2.50	4.12	3.50
	0.6-0.8	-	-	-	3.27	3.25
	0.8-1.0	-	-	-	-	3.00

we generalized an approximation algorithm from [5] by incorporating the 2-dimensional Newton's method with retrograde analysis. The technique was then used to solve simplified versions of two-player Can't Stop. The official version has over 10^{36} components – too many 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. H.-r. Fang. The nature of retrograde analysis for Chinese chess, part I. *ICGA Journal*, 28(2):91–105, 2005.
2. H.-r. Fang. The nature of retrograde analysis for Chinese chess, part II. *ICGA Journal*, 28(3):140–152, 2005.
3. R. Gasser. Solving Nine Men's Morris. *Computational Intelligence*, 12:24–41, 1996.
4. J. Glenn. An optimal strategy for Yahtzee. Technical Report CS-TR-0002, Loyola College, 4501 N. Charles St, Baltimore MD 21210, USA, May 2006.
5. J. Glenn, H.-r. Fang, and C. P. Kruskal. A retrograde approximate algorithm for one-player can't stop. Accepted by CG'06 conference, to appear, 2006.
6. G. Irving, J. Donkers, and J. Uiterwijk. Solving Kalah. *ICGA Journal*, 23(3):139–147, 2000.
7. R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. In H.J. van den Herik, I. S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Games VII*, pages 135–162. University of Limburg, Maastricht. the Netherlands, 1994.
8. J.W. Romein and H.E. Bal. Solving the game of Awari using parallel retrograde analysis. *IEEE Computer*, 36(10):26–33, October 2003.
9. J. Schaeffer, Y. Björnsson, N. Burch, R. Lake, P. Lu, and S. Sutphen. Building the checkers 10-piece endgame databases. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10. Many Games, Many Challenges*, pages 193–210. Kluwer Academic Publishers, Boston, USA, 2004.
10. K. Thompson. Retrograde analysis of certain endgames. *ICCA Journal*, 9(3):131–139, 1986.
11. K. Thompson. 6-piece endgames. *ICCA Journal*, 19(4):215–226, 1996.
12. P. Woodward. Yahtzee: The solution. *Chance*, 16(1):18–22, 2003.
13. R. Wu and D.F. Beal. Fast, memory-efficient retrograde algorithms. *ICGA Journal*, 24(3):147–159, 2001.

Solving 20×20 Puzzles

Aleksander Sadikov and Ivan Bratko

Artificial Intelligence Laboratory,
Faculty of Computer and Information Science, University of Ljubljana
Tržaška 25, 1000 Ljubljana, Slovenia
e-mail: {aleksander.sadikov;ivan.bratko}@fri.uni-lj.si

Abstract. We report that it is possible to solve random instances of the 399 Puzzle, the 20×20 version of the well-known sliding-tile puzzles, using real-time A* search (RTA*) in a reasonable amount of time (several hours on a regular PC). Previously, very few attempts were even made to go beyond the 5×5 puzzles. The discovery is based on a recent finding that RTA* works much better with strictly pessimistic heuristics. Our novel contribution is a strictly pessimistic heuristic for sliding-tile puzzles.

1 Introduction

The sliding-tile puzzles have served as a testbed in the search and heuristic-search fields since as long as one can remember, e.g., in Nilsson's well-known book on search [1] or in Doran and Michie's experiments with the Graph Traverser program [2]. And yet, they are still very much alive and challenging to modern search techniques as demonstrated by the abundance of very recent papers, e.g., [3–5].

The puzzles pose several different interesting problems; two of these are (a) solving them optimally (i.e. in the least amount of moves possible), and (b) solving them at all (with suboptimal solutions). Another nice property of the puzzles is that they are of various sizes — the same problem, yet bigger and bigger in scale. This property is very useful in a testbed as it challenges new algorithms and instills a healthy competition among researchers.

The motivation for our research is our recent discovery that strictly pessimistic heuristic functions enable real-time search to work much better and thus to solve (much) bigger problems [6]. Incomplete-search methods do not guarantee finding an optimal solution even when used in conjunction with admissible heuristics. Thus, the main reason for using admissible and consequently optimistic heuristics is void. As we argued before [6], people nevertheless use optimistic heuristic functions with incomplete-search methods, because: (a) they are often readily available, since they were developed for complete-search algorithms, (b) common sense saying that since they proved useful with complete-search methods, perhaps they are useful with incomplete-search methods as well, and (c) it was never shown that they could be counterproductive.

While we provided a theoretical proof that pessimistic heuristics enable real-time search to make better decisions and backed-up our theoretical deliberations with experiments in the 3×3-puzzle (this small puzzle was used, because we needed to know true values of every legal position to make a complete evaluation of the approach) with artificially constructed heuristics, we left open the question how effective can real-time search be with a real-life pessimistic function. We also did not explain how one is to construct pessimistic heuristics.

In the present paper we were primarily interested in how much bigger problems (if at all) it was possible to solve by using strictly pessimistic heuristics with RTA* as suggested. The quality of solutions was of secondary importance. We managed to construct a strictly pessimistic heuristic by mimicking how humans approach the puzzle and by recursively decomposing it into puzzles of smaller order. We then experimentally tested this heuristic against the Manhattan heuristic on puzzles of various sizes. The results we obtained are quite astonishing!

The paper is organized as follows. First, we give an extended history of research inspired or tested by the sliding-tile puzzles. Then we present the construction sketch for our pessimistic heuristic. In the experimental section we compare it with the Manhattan heuristic on sliding-tile puzzles of various sizes. We conclude with a brief discussion and present some of our future goals.

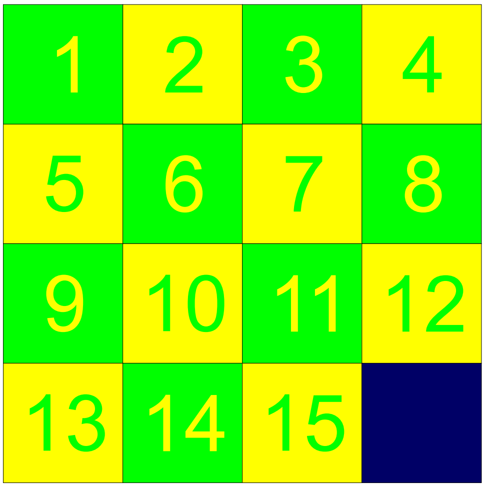


Fig. 1. Fifteen puzzle (goal state).

2 History

The sliding-tile puzzles have always served as a testbed in the search and heuristic-search fields because of their large state-spaces and their scalability (various sizes with similar properties). The state-space size of an N×N-puzzle is (N×N)!/2, meaning that even the modest Fifteen (4×4) puzzle contains more than 10^{13} legal positions. Brute-force search algorithms are thus quickly rendered helpless, and heuristic-search methods called upon.

However, before we completely turn our attention away from brute-force search, we should mention what is possibly the largest breadth-first search in history — Korf and Schultze [3] completely solved the Fifteen puzzle, finding out that the hardest problem takes 80 moves to solve optimally.

A lot of research was geared towards solving the puzzles optimally. The smallest, Eight (3×3) puzzle, containing 181,440 legal positions, could be solved with A* [7]. However, the next bigger puzzle, the Fifteen puzzle shown in Fig. 1 was beyond the capabilities of A*; it ran out of memory. Optimally solving the Fifteen puzzle required the space-saving trait of a new (at that time) algorithm, IDA* [8]. Thus, the next challenge became the Twenty-Four (5×5) puzzle. This time the breakthrough did not come in the form of a new algorithm; instead, it came in the form of an improved admissible heuristic. Previously, the Manhattan distance was the heuristic of choice (the sum of the Manhattan distances of all tiles from their proper positions). The improved heuristic was still based on the Manhattan heuristic, yet took into account some conflicts between tiles. Ten random instances of the Twenty-Four puzzle were solved with IDA* using this new admissible heuristic [9]. This happened already in 1996, and in the subsequent decade the next largest puzzle still remains unsolved optimally. The latest attempts to solve it center around tablebase-like ideas from two-player games, tabulating smaller puzzles and attempting to use these databases to create an even more accurate admissible heuristic [4, 10].

In 1990, Korf [11] introduced real-time search (RTA*). Apart from being able to make decisions in real time, this on-line algorithm was able to solve much larger problems than off-line algorithms could. Making real-time decisions of course meant sacrificing optimality. Yet, in the same paper, Korf also introduced learning real-time search (LRTA*), a learning modification of RTA* that could learn to improve its solution by repetition (on the same problem). He also proved that with admissible heuristics LRTA* would eventually converge to an optimal solution (as the number of repetitions approaches infinity).

It is interesting that of the two algorithms, the latter, LRTA*, became much more popular than the seemingly more practical RTA*. The problem of finding optimal solutions with LRTA* became a new challenge. However, researchers quickly realized that the convergence towards optimal solutions usually takes unreasonable amounts of time. LRTA* today still cannot optimally solve the Fifteen puzzle. Various approaches have been put forward to speed up the convergence process (while maintaining the quest for optimality); most notable attempts include algorithms such as FALCONS [12], HLRTA* [13], and a combination of both called eFALCONS [14]. These algorithms tweaked with value-update rule of LRTA* (HLRTA*), with action-selection rule of LRTA* (FALCONS), or both (eFALCONS). Still, none of these algorithms was able to optimally solve the Fifteen puzzle. Another idea was also to speed up the calculation of heuristics [15].

In view of this, an idea of relaxing the strive for optimality was bound to come along. Shimbo and Ishida [16] proposed going after somewhat suboptimal solutions — while strictly controlling the level of suboptimality. They introduced the idea of weighted LRTA* which assigns weights to admissible heuristics thus making them possibly pessimistic (but only to a certain degree, controlled by the parameter ϵ; this parameter represents the weight). They proved that solutions so obtained are suboptimal by at most the factor of ϵ. In practice, however, the solutions tend to be much better, but not optimal.

It is quite interesting how much effort was spent on finding optimal (or suboptimal to a controlled degree) solutions with LRTA* as opposed to finding *any* solutions with RTA* quickly. After all, since one of the inherent qualities of RTA* is the ability to solve larger problems than off-line search methods could, one would expect this to be the focal point of research.

Yet, perhaps the absence of papers describing research of going after any solution can be explained by a somewhat surprising fact: RTA* with admissible heuristics was not able to scale up to the extent worth reporting. Indeed, as shown later, our experiments with the Manhattan heuristic demonstrate that RTA* cannot go beyond 5×5 puzzles. And since some of these puzzles were already *optimally* solved in the mean time this is clearly not the best advertisement for the usefulness of RTA*.

In 2006 we proposed changing the *nature* of the underlying heuristics [6]. We proposed to run RTA* with pessimistic heuristics to be able to solve bigger problems. How much bigger is the aim of this paper to show.

Of course, there are other uses of real-time search besides solving sliding-tile puzzles. Robotic navigation and map building is one such example. A beautiful overview of real-time search techniques and applications is given by Koenig [17]. But as a testbed the puzzles remain challenging and fun, and a natural source for a good competition which in turn spurs the research further. We hope that this paper will rekindle research into finding suboptimal solutions to very large problems fast by setting a new benchmark.

3 Pessimistic Heuristic for Sliding-Tile Puzzles

There are two main ideas involved in the design of our pessimistic heuristic. The first is modelling it after the way humans usually approach solving the puzzle. The second is a basic problem-solving technique of decomposing a harder (bigger) problem into two or more easier (smaller) ones. The last ingredient is recursion.

The guarantee that the heuristic is strictly pessimistic (i.e. never underestimates) comes from the following fact: since we actually solved[1] it (in a human-like way) it can at best be solved optimally. But most likely it will be solved suboptimally. Therefore, the heuristic is strictly pessimistic.

[1] Our heuristic actually solves the puzzle to obtain an upper bound on the solution cost. However, the idea of the paper is to show that pessimistic evaluation functions outperform optimistic ones when used with real-time search.

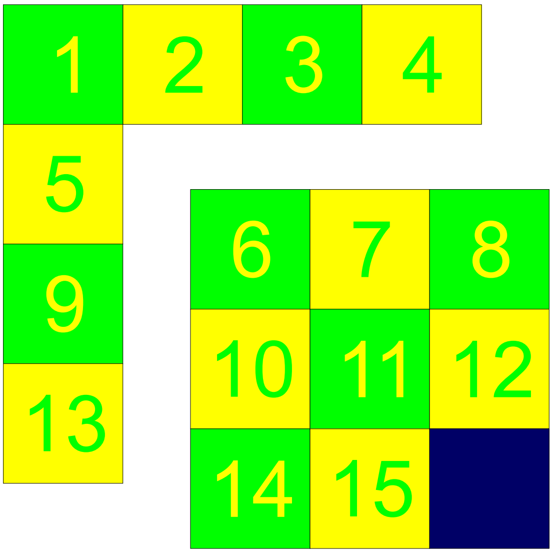


Fig. 2. Decomposition into a smaller puzzle.

The heuristic is based on the decomposition of solving an N×N-puzzle into a partial solution of this puzzle plus the solving of an (N-1)×(N-1)-puzzle. The decomposition step is shown in Fig. 2. Accordingly, the heuristic upper bound on the solution length is computed as the cost of solving the left-most column and the top-most row of the N×N-puzzle, plus (recursively) the heuristic estimate of the cost of the remaining (N-1)×(N-1)-puzzle.

It is possible to solve the top row, and then the leftmost column of the puzzle one tile at a time without compromising the part already solved. A somewhat tricky part is in solving the penultimate and ultimate tiles of the row (column). However, it is possible to do this as follows. The penultimate tile (3 in the case of the Fifteen puzzle in Fig. 2) can be first put in the *last* position of the row, then the last tile (4) just below it (in the second row, the home position of tile 8). The blank is then maneuvered into the home position of tile 3. Then the following sequence of moves achieves the solution of the whole row: "RD", where R means moving the blank right, and D moving the blank down. Analogously, it is possible to solve the leftmost column, the sequence in this case being "DR".

3.1 The Quality of the Heuristic

If one has a perfect or nearly perfect heuristic there is no need to worry about the size of the problem — a direct path (not necessarily optimal, of course) is guaranteed with any reasonable heuristic-search method. Therefore, a relevant question needs to be asked: how good is our pessimistic heuristic? Is it nearly

Table 1. Experimental results for various puzzle sizes. All statistics are per puzzle (averaged over the test set).

	Manhattan			Decomposition		
Puzzle size	#moves	CPU time (s)	Deg. factor	#moves	CPU time (s)	Deg. factor
4×4	7,827.7	1.51	147.55	98.3	0.10	1.85
5×5	339,537.6	354.0	3,309.33	215.8	0.50	2.10
6×6	cannot solve			392.6	1.57	not known
10×10	cannot solve			2,194.6	124.5	not known
15×15	cannot solve			10,927.0	3,353.7	not known
20×20	cannot solve			29,085.5	13,585.1	not known

perfect? We give the answers to these questions by comparing it to the Manhattan heuristic on puzzles for which we know the true values.

The average value of the (optimistic) Manhattan distance heuristic evaluation over the 100 puzzles optimally solved by Korf [8] is 69% of the true solution costs, whereas the average value of (pessimistic) decomposition heuristic evaluation is 250% of the true costs. Of even greater interest is the discrimination power of the two heuristics in deciding which of the two given 4×4 puzzles is easier (has shorter optimal solution). The Manhattan distance correctly decides in 74.2% of all 100×99/2 possible pairs of Korf's 100 4×4 puzzles, whereas the decomposition heuristic only gives correct decision in 56.9% of these pairs.

Thus, we conclude that the decomposition heuristic is actually significantly worse than the Manhattan heuristic by both measures. This result came as quite a surprise to us.

4 Experiments

We compared RTA* search using the Manhattan and our decomposition heuristics on puzzles of increasing size. The Table 1 shows the results obtained.

We measured the solution quality (length), CPU time needed, and degradation factor (ratio between the average cost of solution and average cost of optimal solution). The CPU time is a fairer measure than number of moves, because the two compared heuristics take different times to compute; the decomposition being significantly slower. Tests were run on an average[2] PC and the programs were entirely written in Python scripting language for the simplicity of later making the code available.

The testsets for puzzles of various sizes were as follows. We used Korf's 100 optimally solved puzzles from [8] for 4×4 puzzles, and Korf's 10 optimally solved 5×5 puzzles from [9]. The latter set is small, but gives us the advantage of being able to compute the degradation factor. For puzzles of higher orders we used testsets of ten random positions.

[2] IBM ThinkPad Z60m with 2 GHz processor and 1 GB of RAM.

The most striking result is that while the Manhattan distance could not solve any of the puzzles beyond the 5×5 puzzles, the decomposition heuristic could scale all the way up to 20×20 puzzles, which contain approximately 3.2×10^{868} legal positions! The limit we imposed on the search was 1,000,000 moves made. The depth of lookahead was one-ply (often used by other researchers to keep the planning times low). We have tried other lookahead depths and it did not significantly change the results.

Another important result is also that average solutions generated by the decomposition heuristic are not unreasonably bad (cf. degradation factor). Of course, single-run solutions can be very bad, but on average this is not the case. These solutions are incomparably better than those provided by the Manhattan heuristic.

5 Conclusions and Further Work

We showed that coupling RTA* search with a strictly pessimistic heuristic enabled it to solve drastically bigger problems than with the optimistic, admissible heuristic, even though the latter was of significantly better quality. The results are in line with our previous findings [6], but also show that the extent of the benefit is much bigger than anticipated.

The solutions obtained are far from being terrible and thus useless. In view of the size of the puzzles solved we believe this presents a new benchmark for research focusing on finding suboptimal solutions to large problems.

An intuitive explanation for the obtained results is that the pessimistic heuristic guides the search in the direction of the goal by constantly trying to minimize the "guaranteed" (as much as possible by the quality of the heuristic) upper-bound on the remaining path to the goal. The optimistic heuristic guides the search by providing the lower-bound, in fact encouraging it to look for better and better paths which usually results in a lot of exploration.

Once we get a solution, any solution, we can work on improving it. Not necessarily in the spirit of LRTA*; one can approach this task in other ways (which can, of course, be based on domain-specific knowledge, or whose success can be based on the domain). One such idea is to look for cycles in the obtained solution path and removing them. We feel that this can drastically improve the solutions, at least in domains like mazes and sliding-tile puzzles. This is one avenue of further work we intend to pursue.

The more obvious goal (in the spirit of this paper) to pursue is of course going after even bigger puzzles. While we feel that solving slightly bigger puzzles is already trivial (e.g., the 22×22-puzzle), our next goal is solving the 32×32 or 1,023-puzzle.

Acknowledgements

This work was partly funded by ARRS, Slovenian Research Agency.

References

1. Nilsson, N.: Problem-Solving Methods in Artificial Intelligence. New York: McGraw-Hill (1971)
2. Doran, J.E., Michie, D.: Experiments with the Graph Traverser program. Proceedings of the Royal Society A **294** (1966) 235–259
3. Korf, R.E., Schultze, P.: Large-scale parallel breadth-first search. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05). (2005) 1380–1385
4. Felner, A., Korf, R.E., Hanan, S.: Additive pattern database heuristics. Journal of Artificial Intelligence Research **22** (2004) 279–318
5. Bulitko, V., Lee, G.: Learning in real-time search: A unifying framework. Journal of Artificial Intelligence Research **25** (2006) 119–157
6. Sadikov, A., Bratko, I.: Pessimistic heuristics beat optimistic ones in real-time search. In Brewka, G., ed.: Proceedings of the Seventeenth European Conference on Artificial Intelligence, Riva di Garda, Italy (2006) 148–152
7. 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 Cybernetics **4**(2) (1968) 100–107
8. Korf, R.E.: Depth-first iterative deepening: An optimal admissible tree search. Artificial Intelligence **27**(1) (1985) 97–109
9. Korf, R.E., Taylor, L.A.: Finding optimal solutions to the twenty-four puzzle. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), Portland, OR (1996) 1202–1207
10. Korf, R.E., Felner, A.: Disjoint pattern database heuristics. Artificial Intelligence **134**(1–2) (2002) 9–22
11. Korf, R.E.: Real-time heuristic search. Artificial Intelligence **42**(2-3) (1990) 189–211
12. Furcy, D., Koenig, S.: Speeding up the convergence of real-time search. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence, Menlo Park, CA (2000) 891–897
13. Thorpe, P.: A hybrid learning real-time search algorithm. Master's thesis, Computer Science Department, University of California at Los Angeles (1994)
14. Furcy, D., Koenig, S.: Combining two fast-learning real-time search algorithms yields even faster learning. In: Proceedings of the Sixth European Conference on Planning (ECP-01), Toledo, Spain (2001)
15. Liu, Y., Koenig, S., Furcy, D.: Speeding up the calculation of heuristics for heuristic search-based planning. In: Eighteenth National conference on Artificial Intelligence (AAAI-02). (2002) 484–491
16. Shimbo, M., Ishida, T.: Controlling the learning process of real-time heuristic search. Artificial Intelligence **146**(1) (2003) 1–41
17. Koenig, S.: Agent-centered search. Artificial Intelligence Magazine **22**(4) (2001) 109–131

Reflexive Monte-Carlo Search

Tristan Cazenave

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

Abstract. Reflexive Monte-Carlo search uses the Monte-Carlo search of a given level to improve the search of the upper level. We describe the application to Morpion Solitaire. For the non touching version, reflexive Monte-Carlo search breaks the current record and establishes a new record of 78 moves.

1 Introduction

Monte-Carlo methods have been applied with success to many games. In perfect information games, they are quite successful for the game of Go which has a huge search space [1]. The UCT algorithm [9] in combination to the incremental development of a global search tree has enabled Go programs such as CRAZY STONE [2] and MOGO [11] to be the best on 9×9 boards and to become competitive on 19×19 boards.

Morpion Solitaire is a single-player complete-information game with a huge search space. The current best algorithms for solving Morpion Solitaire are based on Monte-Carlo methods. We present in this paper an improvement of the usual Monte-Carlo method based on reflexivity.

In the next section we present Morpion Solitaire. In the third section we detail reflexive Monte-Carlo search. In the fourth section we give experimental results.

2 Morpion Solitaire

Morpion Solitaire was first published in Science & Vie in April 1974. It has also been addressed in Jeux & Stratégie in 1982 and 1983. The current record of 170 moves has been found by hand at this time.

2.1 Rules of the Game

Figure 1 gives the starting position for Morpion Solitaire. A move consists in adding a circle and drawing a continuous line such that the line contains the new circle as well as four other circles. A line can be horizontal, vertical or diagonal. The goal of the game is to play as many moves as possible.

Figure 2 shows a game after five moves. In this figure only horizontal and vertical moves are displayed, diagonal moves are also allowed when possible.

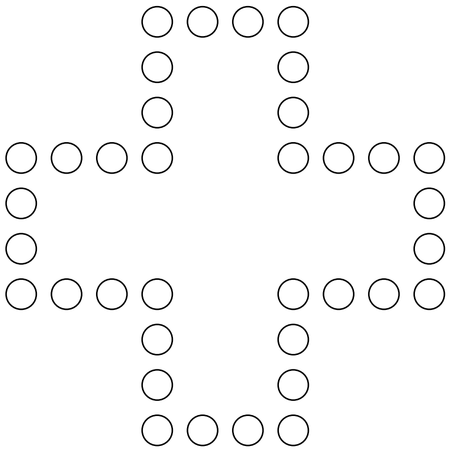


Fig. 1. Initial position.

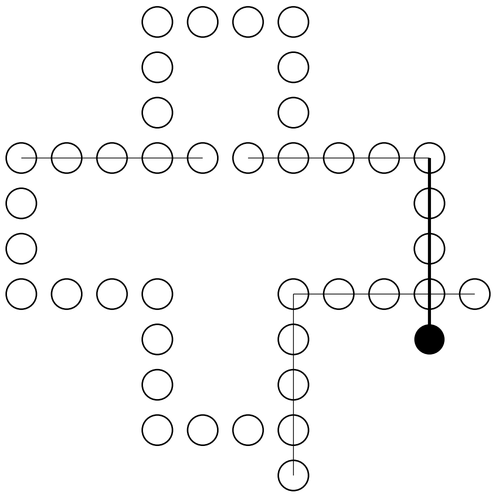


Fig. 2. Position after five moves.

In the original version of the game, two different moves can share a circle at the end of a line even if the lines of the two moves have the same direction. Another version is the non-touching version where two different moves cannot share a circle at the end of a line if their lines have the same direction.

2.2 Previous works on Morpion Solitaire

Hugues Juillé has described an algorithm that finds a 122 moves solution to the touching version [7, 8]. The algorithm was made more incremental by Pascal Zimmer and it reached 147 moves. The 170 moves record has been proved optimal for the last 109 moves [4, 5].

Concerning the non-touching version, the best human record is 68 moves [3]. In his thesis, B. Helmstetter describes a program that finds a 69-moves solution using a retrograde analysis of the 68-moves human record [5]. In December 2006, the best record was 74. It was found with a simulated-annealing algorithm [10, 6]. I found a 76-moves record in December 2006 with Reflexive Monte-Carlo Search, and the 78-moves record in May 2007.

2.3 The Mobility Heuristic

An heuristic for choosing the moves to explore is the *mobility* heuristic. It consists in choosing randomly among the moves that lead to boards that have the maximum number of possible moves. The moves that lead to board that have less possible moves than the boards corresponding to the most mobile moves are discarded.

3 Reflexive Monte-Carlo Search

This section first presents Monte-Carlo search applied to Morpion Solitaire, then it is extended to reflexive Monte-Carlo Search.

3.1 Monte-Carlo search

Monte-Carlo search simply consists in playing random games from the starting position. The code for playing a random game is given in the function $playGame()$:

```
int variation [200];

int playGame () {
  nbMoves = 0;
  for (;;) {
    if (moves.size () == 0)
      break;
    move = chooseRandomMove (moves);
    playMove (move);
    variation [nbMoves] = move;
    updatePossibleMoves (move);
    nbMoves++;
  }
  return nbMoves;
}
```

Moves are coded with an integer, and the $variation$ array contains the sequence of moves of the random game.

The $playGame()$ function is used to find the best game among many random games with the function $findBestMove()$:

```
int nbPrefix = 0;
int prefix [200];
int previousBestScore = 0;
int bestVariation [200];

int findBestMove () {
  int nb = 0, best = previousBestScore - 1;
  for (int i = 0; i < best; i++)
    bestVariation [i] = bestVariation [i + 1];

  initialBoard ();
  for (int i = 0; i < nbPrefix; i++)
     playMove (prefix [i]);
  memorizeBoard ();
  while (nb < nbGames) {
    retrieveBoard ();
    if (moves.size () == 0)
      return -1;
    int nbMoves = playGame ();
    if (nbMoves > best) {
      best = nbMoves;
      for (int i = 0; i < best; i++)
        bestVariation [i] = variation [i];
    }
  }
  previousBestScore = best;
  return bestVariation [0];
}
```

The $prefix$ array contains the moves that have to be played before the random game begins. In the case of a simple Monte-Carlo search, $nbPrefix$ equals zero. In the case of a Meta-Monte-Carlo search the prefix contains the moves of the current meta-game.

The $initialBoard()$ function puts the board at the initial position of Morpion Solitaire. The $memorizeBoard()$ function stores the current board with the list of possible moves, and the $retrieveBoard()$ function restores the board and the list of possible moves that have been stored in the $memorizeBoard()$ function.

3.2 Meta Monte-Carlo Search

Reflexive Monte-Carlo search consists in using the results from Monte-Carlo search to improve Monte-Carlo search. A Meta-Monte-Carlo search uses a Monte-Carlo search to find the move to play at each step of a meta-game. A Meta-Meta-Monte-Carlo search uses a Meta-Monte-Carlo search to find a move at each step of a meta-meta-game, and so on for upper meta levels.

We give below the $playMetaGame()$ function which plays a meta-game:

```
int playMetaGame () {
  previousBestScore = 0;
  for (;;) {
    int move = findBestMove ();
    if (move == -1)
      break;
    prefix [nbPrefix] = move;
    nbPrefix++;
  }
  return nbPrefix;
}
```

This function is used to find the best meta-move in a Meta-Meta-Monte-Carlo search. We give below the code for the $findBestMetaMove()$ function that finds the best meta-move:

```
int nbMetaPrefix = 0;
int metaPrefix [200];
int previousMetaBestScore = 0;
int bestMetaVariation [200];

int findBestMetaMove () {
  int nb = 0, best = previousMetaBestScore - 1;
  for (int i = 0; i < best; i++)
    bestMetaVariation [i] = bestMetaVariation [i + 1];
  while (nb < nbMetaGames) {
    for (int i = 0; i < nbMetaPrefix; i++)
      prefix [i] = metaPrefix [i];
    nbPrefix = nbMetaPrefix;
    int nbMoves = playMetaGame ();
    if (nbMoves - nbMetaPrefix > best) {
      best = nbMoves - nbMetaPrefix;
      for (int i = 0; i < best; i++)
        bestMetaVariation [i] = prefix [nbMetaPrefix + i];
    }
    nb++;
  }
  previousMetaBestScore = best;
  if (best <= 0)
    return -1;
  return bestMetaVariation [0];
}
```

The code for the meta-meta level is similar to the code for the meta-level. In fact all the meta-levels above the ground level use a very similar code. There is no theoretical limitation to the number of levels of reflexive Monte-Carlo search. The only limitation is that each level takes much more time than its predecessor.

4 Experimental Results

For the experiments, the programs run on a Pentium 4 cadenced at 2.8 GHz with 1 GB of RAM. All experiments concern the non-touching version.

Table 1 gives the lengths of the best games obtained with sampling and random moves for different numbers of random games. It corresponds to a simple Monte-Carlo search.

Table 1. Best lengths for different numbers of games with sampling and random moves.

games	10	100	1,000	10,000	100,000	1,000,000	10,000,000
length	55	58	60	63	63	63	64
time	0.008s.	0.12s.	1.0s.	9.8s.	98s.	978s.	17,141s.

Table 2 gives the lengths of the best games obtained with sampling and the *mobility* heuristic for choosing moves in the random games. We can observe that for a similar number of random games, the *mobility* heuristic finds longer games than the completely random choices. However, it also takes more time, and for equivalent search times, the two heuristic find similar lengths.

Table 2. Best lengths for different numbers of games with sampling and mobility.

games	10	100	1,000	10,000	100,000	1,000,000
length	60	61	62	62	63	64
time	0.12s.	1.05s.	10.2s.	101s.	1,005s.	10,063s.

The UCT algorithm uses the formula $\mu_i + \sqrt{\frac{log(t)}{C \times s}}$ to explore moves of the global search tree, μ_i is the average of the games that start with the corresponding move, t is the number of games that have been played at the node, and s is the number of games that have been played below the node starting with the corresponding move. We have tested different values for the C constant, and the best results were obtained with $C = 0.1$.

Table 3 gives the best lengths obtained with UCT for different numbers of simulations. The results are slightly better than with sampling.

Table 3. Best lengths for different numbers of games with UCT and mobility.

games	10	100	1,000	10,000	100,000	1,000,000
length	60	60	61	64	64	65
time	0.16s.	1.3s.	16s.	176s.	2,108s.	18,060s.

Reflexive Monte-Carlo search has been tested for different combinations of games and meta-games, each combination corresponds to a single meta-meta-game. The length of the best sequence for different combinations is given in Table 4. The first line gives the number of meta-games, and the first column gives the number of games. Table 5 gives the time for each combination to complete its search.

Table 4. Best lengths for different numbers of games and meta-games.

	1	10	100	1000
1	63	63	67	67
10	63	67	69	67
100	65	69	72	
1000	71			

Table 5. Times for different numbers of games and meta-games.

	1	10	100	1,000
1	5s.	42s.	417s.	3,248s.
10	23s.	774s.	3,643s.	27,688s.
100	253s.	4,079s.	26,225s.	
1,000	3,188s.			

We have obtained sequences of length 76 with different configurations of the reflexive search. The configuration that found the 78 moves record consists in playing 100 random games at the ground level, 1,000 meta-games at the meta level, and one game at the meta meta level. The search took 393,806 seconds to complete. The sequence obtained is given in figure 3.

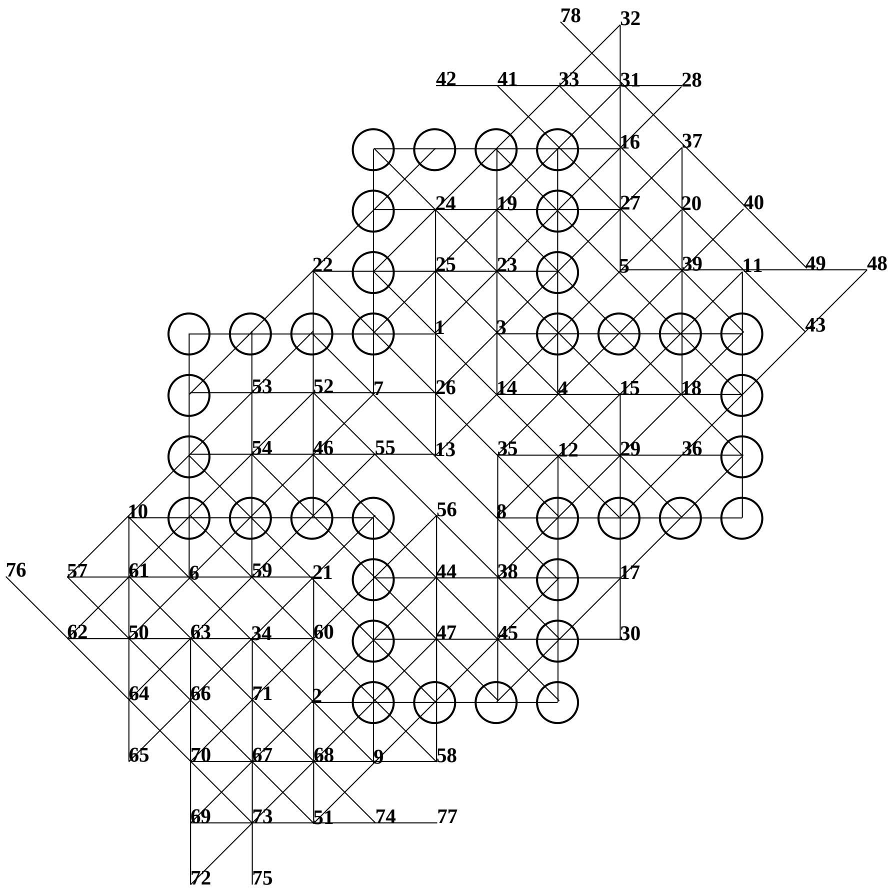


Fig. 3. 78 moves for the non-touching version.

5 Conclusion and Future Research

Reflexive Monte-Carlo search has established a new record of 78 moves for the non-touching version of Morpion Solitaire. This is four moves more than the best attempt made by other programs, and it is ten moves more than the human record for the game. Concerning the touching version the human record of 170 is still much better than what current programs can achieve.

Reflexive Monte-Carlo Search is a general technique that can be applied in other games. Also, future works include the parallelization of the algorithm and its application to other games.

References

1. Bouzy, B., Cazenave, T.: Computer Go: An AI-Oriented Survey. Artificial Intelligence **132**(1) (2001) 39–103
2. Coulom, R.: Efficient selectivity and back-up operators in monte-carlo tree search. In: Proceedings Computers and Games 2006, Torino, Italy (2006)
3. Demaine, E.D., Demaine, M.L., Langerman, A., Langerman, S.: Morpion solitaire. Theory Comput. Syst. **39**(3) (2006) 439–453
4. Helmstetter, B., Cazenave, T.: Incremental transpositions. In H. Jaap Herik, Yngvi Bjornsson, N.S.N., ed.: Computers and Games: 4th International Conference, CG 2004. Volume 3846 of LNCS, Ramat-Gan, Israel, Springer-Verlag (2006) 220–231
5. Helmstetter, B.: Analyses de dépendances et méthodes de Monte-Carlo dans les jeux de réflexion. Phd thesis, Université Paris 8 (2007)
6. Hyyro, H., Poranen, T.: New heuristics for morpion solitaire. Technical report (2007)
7. Juillé, H.: Incremental co-evolution of organisms: A new approach for optimization and discovery of strategies. In: ECAL. Volume 929 of Lecture Notes in Computer Science. (1995) 246–260
8. Juillé, H.: Methods for statistical inference: extending the evolutionary computation paradigm. Phd thesis, Brandeis University (1999)
9. Kocsis, L., Szepesvari, C.: Bandit based monte-carlo planning. In: ECML-06. (2006)
10. Langerman, S.: Morpion solitaire. Web page, http://slef.org/jeu/ (2007)
11. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo go. In: CIG 2007, Honolulu, USA (2007) 175–182

Monte-Carlo Tree Search in Backgammon

François Van Lishout[1], Guillaume Chaslot[2], and Jos W.H.M. Uiterwijk[2]

[1] University of Liège, Montefiore Institute, B28, 4000 Liège, Belgium, `vanlishout@montefiore.ulg.ac.be`
[2] Universiteit Maastricht, Maastricht ICT Competence Center (MICC), Maastricht, The Netherlands, {`G.Chaslot, uiterwijk`}`@micc.unimaas.nl`

Abstract. Monte-Carlo Tree Search is a new method which has been applied successfully to many games. However, it has never been tested on two-player perfect-information games with a chance factor. Backgammon is the reference game of this category. Today's best Backgammon programs are based on reinforcement learning and are stronger than the best human players. These programs have played millions of offline games to learn to evaluate a position. Our approach consists rather in playing online simulated games to learn how to play correctly in the current position.

1 Introduction

Monte-Carlo Tree Search has among others been applied successfully to the game of Go [5, 7], the Production Management Problem [4], the game of Clobber [9], Amazons [10], and the Sailing Domain [9]. These games are either one-player games with a chance factor or two-player games without a chance factor. It seems thus natural to try this approach on a two-player game with a chance factor. The game of Backgammon is the most famous representative of this category and is therefore a good testbed. Furthermore, thanks to the works of Tesauro, it has already been proved that a very strong AI can be written for this game [11]. The actual approach is to train a neural network by playing millions of offline games. In this paper we present a contrasting approach consisting of using only online simulated games from the current position.

2 The Game of Backgammon

Backgammon is a board game for two players in which checkers are moved according to the roll of two dice. The objective of the game is to remove all of one's own checkers from the board. Each side of the board is composed of 12 triangles, called points. Figure 1 gives the initial position from Black's point of view. The points are numbered from 1 to 24 and connected across the left edge of the board. The white checkers move in the clockwise direction and the black checkers in the opposite one. The points 1 to 6 are called Black's *home board* and the points 19 to 24 White's home board.

2.1 Rules of the game

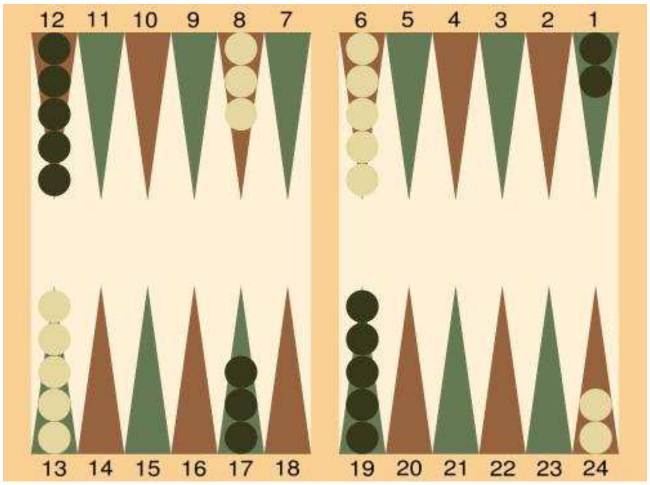


Fig. 1. The initial position of a Backgammon game.

Initially, each player rolls one die and the player with the higher number moves first using both dice. The players then alternately move, rolling two dice at the beginning of each turn. Let's call x and y the number of pips shown on each dice. If possible, the player must move one checker x points forward and one checker y points forward. The same checker may be used twice. If it is impossible to use the dice, the player passes its turn. If $x = y$ (doubles), each die must be used twice.

It is forbidden to move a checker to a point occupied by more than one opponent's checker (the point is *blocked*). If a checker lands on a point occupied by exactly one opponent's checker, the latter is taken and placed on the middle bar of the board. As long as a player has checkers on the bar, he cannot play with his other checkers. To free them, he must play a die which allows him to reach a point of the opponent's home board that is not blocked (e.g., a die of 1 permits to reach the 1-point/24-point and a die of 6 the 6-point/19-point).

When all the checkers of a player have reached the home board, he can start removing them from the board. This is called *bearing off*. A checker can only go out if the die number is the exact number of points to go out. However, there is an exception: a die may be used to bear off checkers using less points if no other checkers can be moved with the exact number of points. If a player has not borne off any checker by the time his opponent has borne off all, he has lost a gammon which counts for double a normal loss. If the losing player still has checkers on the bar or in its opponent's home board, he has lost a backgammon which counts for triple a normal loss [2].

Before rolling the dice on his turn, a player may demand that the game be played for twice the current stakes. The opponent must either accept the new stakes or resign the game immediately. Thereafter, the right to redouble belongs exclusively to the player who last accepted a double.

2.2 Efficient Representation

The choice of an efficient representation for a backgammon configuration is not as trivial as it seems. It must be efficient to do the following operations:

1. determine the content of a particular point,
2. compute where the white/black checkers are,
3. decide if a checker can move from point A to point B,
4. compute if all the white/black checkers are in the home board, and
5. actualize the representation after a move from point A to point B.

The straight-forward idea of using simply an array which gives the number of checkers present in each point is not sufficient. This representation has in fact the drawback that the second and fourth operations require to traverse all the array. Even if the latter has only a size of 24, those operations will be performed millions of time and should go faster. For this reason, we use:

– An array $PointsContent[25]$, where the i^{th} cell indicates how many checkers are on the i^{th} point. $+x$ indicates that x black checkers are present, $-x$ that x white checkers are present, and 0 that the point is empty.
– A vector $BlackCheckers$ whose first element is the number of points containing black checkers and the other elements are the corresponding point number in **increasing** order.
– A vector $WhiteCheckers$ whose first element is the number of points containing white checkers and the other elements are the corresponding point number in **decreasing** order.
– Two integers $BlackCheckersTaken$ and $WhiteCheckersTaken$ which gives the amount of black/white checkers taken.

The first operation can be performed immediately as $PointsContent[i]$ gives the content of point i directly. The second operation takes a time proportional to the length of $BlackCheckers/WhiteCheckers$. The third operation is trivial as it depends only on whether $PointsContent[B]$ is blocked or not. The fourth operation is also immediate: all the black checkers are in the home board if and only if $BlackCheckers[1]$ [3] is greater than 18, and all the white checkers are in the home board if and only if $WhiteCheckers[1]$ is smaller than 7.

The fifth operation requires to adapt $PointsContent[A]$ and $PointsContent[B]$ to their new content which is immediate. The point A must be removed from $Blackcheckers/WhiteCheckers$ if A contains only one own checker. The point B must be added if it was empty or contained one opponent's checker. In this latter case, it must be removed from $WhiteCheckers/BlackCheckers$.

[3] Since $BlackCheckers$ is sorted in increasing order from the second element on, the element indexed 1 is the lowest point.

3 Monte-Carlo Tree Search

In this section, we describe the structure of an algorithm based on Monte-Carlo tree search. This is a standard Monte-Carlo Tree Search [7] adapted to include chance nodes.

3.1 Structure of MCTS

In MCTS, a node i contains at least the following two variables: (1) the value v_i (usually the average of the results of the simulated games that visited this node), and (2) the visit count n_i. MCTS usually starts with a tree containing only the root node. We distinguish two kinds of nodes. *Choice nodes* correspond to positions in which one of the players has to make a choice. *Chance nodes* correspond to positions in which dice are rolled.

Monte-Carlo Tree Search consists of four steps, repeated as long as there is time left. (1) The tree is traversed from the root node to a leaf node (L), using a *selection strategy.* (2) An *expansion strategy* is called to store one (or more) children of L in the tree. (3) A *simulation strategy* plays moves in self-play until the end of the game is reached. (4) The result R of this "simulated" game is $+1$ in case of a win for Black, and -1 in case of a win for White. R is backpropagated in the tree according to a *backpropagation strategy.* This mechanism is outlined in Figure 2. The four steps of MCTS are explained in more details below.

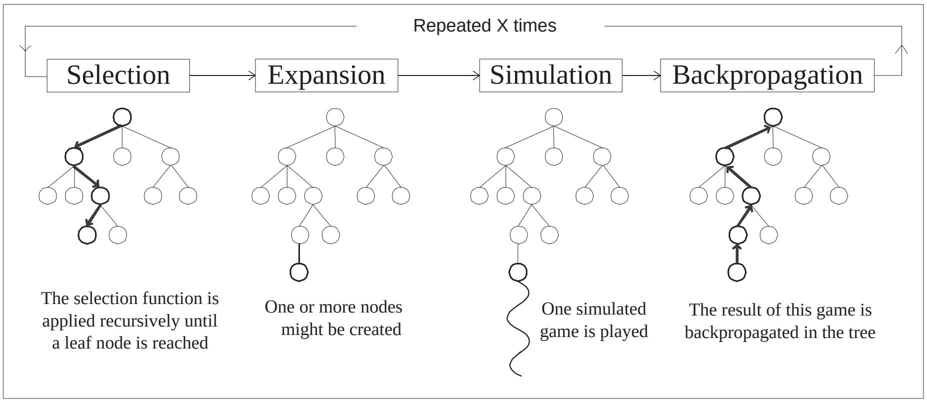

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes might be created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 2. Scheme of a Monte-Carlo Tree Search.

Selection is the strategic task that selects one of the children of a given node. In chance nodes, the next move will always be chosen randomly, whereas in a choice node it controls the balance between exploitation and exploration.

On the one hand, the task is often to select the move that leads to the best results (exploitation). On the other hand, the least promising moves still have to be explored, due to the uncertainty of the evaluation (exploration). This problem is similar to the Multi-Armed Bandit problem [6]. As an example, we mention hereby the strategy UCT [9] (UCT stands for Upper Confidence bound applied to Trees). This strategy is easy to implement, and used in many programs. The essence is choosing the move i which maximizes formula 1:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}} \tag{1}$$

where v_i is the value of the node i, n_i is the visit count of i, and N is the visit count of the parent node of i. C is a coefficient, which has to be tuned experimentally.

Expansion is the strategic task that, for a given leaf node L, decides whether this node will be expanded by storing some of its children in memory. The simplest rule, proposed by [7], is to expand one node per simulation. The node expanded corresponds to the first position encountered that was not stored yet.

Simulation (also called **playout**) is the strategic task that selects moves in self-play until the end of the game. This task might consist of playing plain random moves or – better – pseudo-random moves. Indeed, the use of an adequate simulation strategy has been shown to improve the level of play significantly [3, 8]. The main idea is to play better moves by using patterns, capture considerations, and proximity to the last move.

Backpropagation is the procedure which backpropagates the *result* of a simulated game (win/loss) to the nodes it had to traverse to reach the leaf. The *value* v_i of a node is computed by taking the average of the results of all simulated games made through this node. This procedure affects equally chance nodes and choice nodes.

Finally, the move played by the program is the child of the root with the highest visit count.

4 The Computation of the Online Random Games

Until today, computing the random games very fast was not really an issue. Since the computation was made offline, winning some microseconds was not that important. The difficult part in playing random games is to be able to compute the list of all the possible moves efficiently. Once this is done, a simple program can be used. It should choose dice values at random, call the functions that find all the possible moves and choose one of them randomly, and repeat for both players until the end of the game is reached.

4.1 Computing all the Possible Moves

When the dice are not doubles, the number of possible moves is small and easy to compute. Even if we do have doubles only once every six dice rolls, the number of possible moves will be so much higher that it is very important to take care of computing this very fast.

Let us define a *tower* as a point containing own checkers. The straightforward idea for doubles in many programs is to use a simple loop which traverses the board to find a tower from which it is possible to make a first move. Then, another loop is embedded to find all the possible ways to make a second move from either the same tower or another one not already traversed by the main loop. And so on for the third and fourth move.

The obvious problem of this method is that the embedded loops consume time by traversing the same parts of the board again and again. A less obvious problem of this method is that it generates clones when the same position can be obtained through different moves. They must then be deleted afterwards.

Therefore, we have used a software-engineering approach for solving this problem. We have isolated a simple subproblem and solved it efficiently. Then, we have added more and more features to this subproblem to finally be able to solve the original question. The idea is to compute how often each tower can be used maximally. This depends on the number of checkers of the tower, and of the maximum number of steps that can be done from this tower with the same checker.

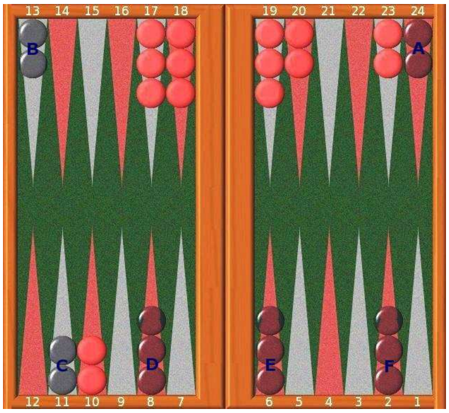


Fig. 3. A typical backgammon position. Black has six towers labelled from A to F.

For example, in the position of figure 3, suppose that it is Black's turn and that the dice are 3-3. The towers B and F cannot be used at all, because the 10-point is blocked and because it is not allowed to start the bearing off already.

Tower A can be used maximally two times since each checker can be moved to the 21-point, but none can go to the 18-point which is blocked. Tower E can be used maximally three times since each checker can be used to move one step. Finally, all the dice could be used on the towers C and D so that they can be used maximally four times. This gives us, if we ignore the towers that cannot be used, the following table of the possible towers uses:

A	C	D	E
2	4	4	3

Our primitive function takes as argument this table and the number of dice that will be used. From this, it computes all the possible ways to use the towers. In our example, it would be the following list:

$AACC, AACD, AACE, AADD, AADE, AAEE, ACCC, ..., DEEE$

The idea of our algorithm is recursive. One basic case is when only one die must be used. In this case, we return the list of all the towers that can be used at least once. The second basic case is when the table is empty. An empty list is returned as there is no tower to use at all. In the other cases, we separate the solutions in two complementary groups:

– The solutions for which the first tower of the table is not used. These are computed by a recursive call where the first entry of the table has been removed and the number of dice has been kept.

– The solutions for which the first tower of the table is used at least once. These are computed by a recursive call where the table has been kept and the number of dice has been decreased by one. Adding one more use of the first tower to all these solutions solves the problem.

Now that we know all the possible towers uses, we still do not know all the possible moves. Indeed, using a tower three times for example can be executed in different ways: move the upper checker of the tower three steps ahead, make two steps with the upper checker and one step with the second checker, or make three one-step moves with three checkers. Note that making one step with the upper checker and then two steps with the second is not another possibility. It leads to the same position as the second possibility.

The second step of our engineering approach was thus to extend the solution found by our primitive function to all the possible real moves. This can be achieved very fast. The idea is again recursive. If the list of the tower moves is empty, the list of the real moves is also empty. Else, we compute all the possible real moves corresponding to the first tower move of the list. Then we compute recursively all the other possible real moves and add the new ones in front.

The computation of all the possible real moves corresponding to a particular tower move has been precomputed manually. This means that the computer does not have to waste time by computing them using an algorithm, because we have coded all the elementary cases one by one. The code is thus a long cascade of *if..then..else* statements, but it executes extremely fast.

Now we have a function that finds all the possible moves when we are free to move as we want. However, when some of our own checkers are taken, we are not. We must first reenter the checkers from the bar and can only then choose the remaining moves freely. As we are in the case of doubles, deciding if we can reenter checkers at all or not is trivial. It is possible if and only if the x-th point of the opponent's home board is not blocked, where x is the dice value. And if it is possible to reenter one, it is also possible to reenter two, three or four (on the same point).

5 McGammon 1.0

Our program has been called McGammon, where MC of course stands for Monte Carlo. In this paper we present the very first version of the program. We have made a simplification of the game rules to compute the random games faster and already have early results. The simplification is that when all the checkers are in the home board, the player directly wins the game.

Our program plays thus bad at the approach of the endgame, where it tries to accumulate the checkers on the left-part of the home board. We are currently working on another version which implements good bearing-off strategies. We expect thus this problem to disappear in our next version. The new version will be finished soon and will compete in the Olympiad 2007 competition.

5.1 Results

The program is able to play about 6500 games per second on a quad-opteron 2.6 GHz. As a test-bed, we have used the starting position of the game. For the fifteen possible initial dice (no doubles are allowed for the first move), we have run 200,000 random games to choose our first move. We have compared the moves found with the suggestions given online by professional players [1], see Table 1.

Dice roll	McGammon move	% Expert playing this move	Main move played by the experts
1-2	8/6, 24/23	0%	13/11, 24/23 (60.1%)
1-3	8/5, 6/5	99.9%	8/5, 6/5 (99.9%)
1-4	13/8	2.2%	24/23, 13/9 (74.5%)
1-5	24/23, 13/8	72.8%	24/23, 13/8 (72.8%)
1-6	13/6	0%	13/7, 8/7 (99.8%)
2-3	13/10, 8/6	0%	24/21, 13/11 (51.8%)
2-4	13/9, 8/6	0%	8/4, 6/4 (99.8%)
2-5	13/8, 8/6	0%	13/11, 13/8 (55.4%)
2-6	8/2, 8/6	0%	24/18, 13/11 (81.4%)
3-4	13/6	0%	24/20, 13/10 (38.8%)
3-5	8/3, 6/3	97.8%	8/3, 6/3 (97.8%)
3-6	13/4	0%	24/18, 13/10 (76.4%)
4-5	13/9, 13/8	30.5%	24/20, 13/8 (63.1%)
4-6	8/2, 6/2	27.3%	24/18, 13/9 (37.0%)
5-6	13-2	0%	24/13 (99.3%)

Table 1. Opening moves played by McGammon

Our program has found three strong advises and two openings played by around 30% of the professionals. It has also found one move played by a small minority of experts. For the other nine initial dice, McGammon plays very strange moves, especially for 1-6 and 2-4 where most of the professional players agree on another move than the one found by our program. This is due to the fact that we work with simplified rules and the program tries to put checkers in the home board as soon as possible because it thinks that this is the way to win a game. This explains why it plays a checker on the 6-point as soon as it can. We expect this kind of moves to disappear in our next version.

6 Conclusion

In this paper, we have shown the first results of a Monte-Carlo tree search in the game of Backgammon. Even if we have only implemented a simplification of the game, our program is already able to find some expert moves. Despite of the poor quality of the random games, the games played with good starting moves achieve more wins than the ones played from obvious bad moves.

As future work, we will very soon finish the implementation of the bearing-off strategies. Then, we plan to add human-expert strategies to the random games to improve their quality. In the long run, it could also be envisaged to combine learning and Monte-Carlo Tree Search. One possible idea is to learn strategies offline and use them in our random games. A more promising idea is to play random games with a program from the learning community but pilot them with the Monte-Carlo Tree Search algorithm.

References

1. http://en.wikipedia.org/wiki/backgammon.
2. http://www.bkgm.com/openings.html.
3. Bruno Bouzy. Associating Domain-Dependent Knowledge and Monte-Carlo Approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, 175(4):247–257, 2005.
4. Guillaume Chaslot, Steven De Jong, Jahn-Takeshi Saito, and Jos W. H. M. Uiterwijk. Monte-Carlo Tree Search in Production Management Problems. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 91–98, 2006.
5. Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, Jos W. H. M. Uiterwijk, and H. Jaap van den Herik. Monte-Carlo Strategies for Computer Go. In Pierre-Yves Schobbens, Wim Vanhoof, and Gabriel Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.
6. Pierre-Arnaud Coquelin and Rémi Munos. Bandit Algorithm for Tree Search. Technical Report 6141, INRIA, 2007.
7. Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proceedings of the 5th Computers and Games Conference*, 2007 (in press).
8. Sylvain Gelly, Yizao Wang, Remi Munos, and Olivier Teytaud. Modifications of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
9. Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In *ECML-06, LNCS/LNAI 4212*, pages 282–293, 2006.
10. Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo Search. In *White paper*, 2006. http://zaphod.aml.sztaki.hu/papers/cg06-ext.pdf.
11. Gerald Tesauro. Temporal difference learning and TD-gammon. *Communications of the ACM*, 38(3), 1995.

The Monte-Carlo Approach in Amazons

Julien Kloetzer[1], Hiroyuki Iida[1], and Bruno Bouzy[2]

[1] Research Unit for Computers and Games
Japan Advanced Institute of Science and Technology
[2] Centre de Recherche en Informatique de Paris 5
Université René Descartes

Abstract. The game of the Amazons is a quite new game whose rules stand between the game of Go and Chess. Its main difficulty in terms of game programming is the huge branching factor. Monte-Carlo is a method used in game programming which allows us to overcome easily this problem. This paper presents how the Monte-Carlo method can be best adapted to Amazon programming to obtain a good level program, and improvements that can be added to it.

1 Introduction

The game of Amazons (in Spanish, *El Juego de las Amazonas*) has been invented in 1988 by Walter Zamkauskas of Argentina. Although it is very young, it is already considered as being a difficult game: its complexity is between Chess and the game of Go.

The main difficulty of the game of Amazons is its complexity: the average number of moves is 80, with a branching factor of approximately 500, and no more than 2176 available moves for the first player. With this in head, we can clearly see that an exhaustive full tree-search is a difficult task: there are no more than 4 millions different positions after two moves. Moreover, even if many of theses moves are clearly bad, there are often positions where more than 20 moves can be considered as “good” moves, and selecting these moves is a hard task [2].

Monte-Carlo (short: MC) is a simple game-independent algorithm which has recently proven to be competitive for the game of Go, especially including a part of Tree Search (different from the classical minimax approach), and knowledge of the game. Now, considering that the game of Go has the same main drawback as the Amazons, a huge branching factor, and that both these games are territory-based games, thus increasing the similarity, we can expect the Monte-Carlo approach to give good results for the game of Amazons.

After a brief description of the rules of the game, section 2 of this paper focuses on the main core of a Monte-Carlo Amazons program. Section 3 presents some improvements that can be added to it, with the results discussed in section 4. Section 5 focuses on our latest improvement to the program, before concluding and discussing future works in section 6.

2 Using Monte-Carlo for the Game of Amazons

The rules of this game (usually called simply "Amazons") are very simple: it is played on a square board of 10x10, sometimes less but this size is the classical one. Each player begins with 4 Amazons, placed all over the board (left of figure 1). A player move consists first on moving one of his Amazons: it can be moved in every direction in a straight line, on any square accessible from the Amazon, exactly as Queen in Chess. Then, after having moved an Amazon, the player chooses a square accessible from the one on which his Amazon just landed, and *shoots an arrow* on this square: this square becomes blocked until the end of the game. No further Amazon move or arrow shot can go through it or land on it (right of figure 1).

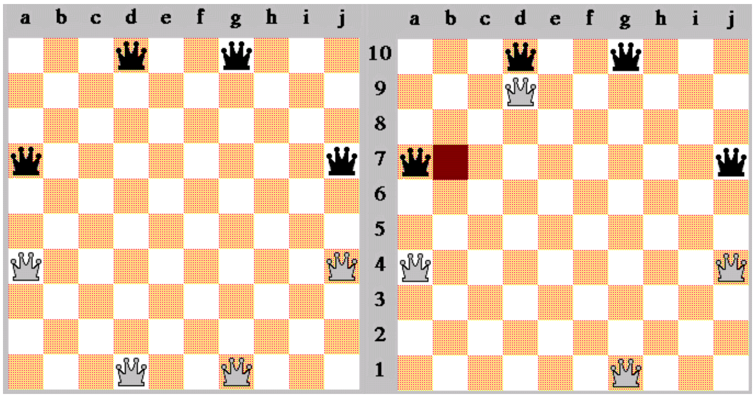


Fig. 1. Beginning position in Amazons (left) and after one move (right).

Each player alternatively makes one of these moves, so a square is blocked on each move. The first player that cannot move any more loses the game, and the score of the game is usually determined as being the number of moves that the other player could have played after it.

All Monte-Carlo Amazons programs should include recent developments made to combine MC and Tree-Search (short: TS). A pseudo code is given in figure 2.

One run (or playout) of the evaluation consists on three steps:

- First, after the search tree used in the evaluation has been initialized to the root itself only (line 2), a new node of the game tree is selected to be added later to the current search tree (line 6), combined with its evaluation.
- Then, a completely random game is performed, starting from the given node, until the end of the game (line 5). The evaluation given is usually either the score of the game (if available), or the information win/loss/draw.

```
 1. Function find_move ( position )
 2. treeSearch_tree = tree ( position )
 3. while ( time is remaining )
 4.   Node new_node = search_new_node ( treeSearch_tree )
 5.   v = evaluate ( new_node )
 6.   treeSearch.add ( v, new_node )
 7. return best_move()
 8.
 9. Function search_new_node ( tree )
10. node = root ( tree )
11. while ( number_unvisited_children ( node ) != 0 )
12.   node = find_best_children ( node )
13. node = random_unvisited_children ( node )
14. return node
```

Fig. 2. Pseudo-code for Monte-Carlo using tree-search.

- Finally, the given node is added to the current search tree: the evaluation given is stored in all the nodes that have been visited in the function search_new_node and in the new node.

The search function is the main core of the tree-search Monte-Carlo model. In previous versions of MC, it always returned one of the children node of the root, either chosen randomly, or so that each node is visited the same number of time [1], or chosen by some knowledge [4]. Following the algorithm UCT [7], our program visits the current search tree by exploring nodes that maximizes the score given by the formula:

$$\text{Score(node(i))} = \text{Evaluation(node}(i)) + \text{C} * \sqrt{\frac{ln(nbVisits(parent(node(i))))}{nbVisits(node(i))}} \quad (1)$$

The first term of formula (1), the evaluation, is usually given either by the expected win/loss ratio of the node, or the expected average score. The number of visits of one node is the number of times it was selected in the process of searching for a new node (line 9 of figure 2). Both these values (number of visits and average evaluation) are updated after each playout (line 6). The second term allows nodes not to be forgotten during the process, raising if the node is not visited while his parent is. Finally, the factor C has to be tuned experimentally.

In the field of Amazons playing, some features need to be discussed:

- Each move consists of two actions (Amazon movement + arrow shot), and usually, one node results from a combination of these actions. However, we can also choose to split every action decision in two: in every random game, an Amazon movement is selected at random, and then an arrow shot from this Amazon, not a combination. Also, we can do the same with the search

tree used by UCT, by not using a usual two levels tree, but a four levels one. On the first level are the positions obtained after an Amazon movement, on the second the positions obtained after an arrow shot from the Amazon just moved, and similarly for the third and fourth level, with movements and shot from the other player.
This changes should allows us basically to run more playouts, and thus to increase in a very simple way the level of the program, because it does not have to compute every move at each position in the random games.
- The evaluation has also to be chosen accordingly to the game which is played. A Win-Loss ratio is usually used for MC + TS in game programming [6], but we could also use an average score, or a combination of both.

Tests and discussion of these three features (splitting in the random games, splitting in the tree used by UCT and the evaluation) are given in section 4.

3 Improving the Random Games

Our program (Campya) includes the algorithm presented in section 2 to choose a move. However, at this state, it lacks seriously of some knowledge of the game, and can easily be defeated by a fair human player.

Improvements to a Monte-Carlo with Tree-search program can basically be made at three levels:

- In the random games, by adding knowledge to it [3]
- In the tree-search part, changing the behaviour of UCT or using other techniques [5]
- At the final decision of the move, for example by pruning moves [4]

Improving the random games has already proven to be a good way to improve the level of a Monte-Carlo program, by adding knowledge to create pseudo-random games. Moves can be chosen more or less frequently according to patterns or to simple rules, as we did here for the game of Amazons. We decided to focus on this method to improve the level of Campya.

3.1 The Liberty Rule

Mobility is an important factor in Amazons. Having an Amazon which is completely or almost completely enclosed at the beginning of the game is like fighting at 3 against 4 for the remaining game. We defined the number of liberties of an Amazon as the number of empty squares adjacent to this Amazon, using a concept similar to the game of Go. Then, we added the following rules to the random games:

- Any Amazon with 1 or 2 liberties has to be moved immediately
- Any opponent's Amazon with 1 or 2 liberties should be enclosed if possible

Two liberties is a critical number: if they are adjacent, one can move an Amazon on one of these, and shoot an arrow on the other one. This way, we punish bad moves, and try to avoid being punished.

3.2 Pruning Moves from Enclosed Amazons

We say that an Amazon is isolated if any of the squares accessible from this Amazon in any number of moves cannot be accessed by opponent's Amazons. An isolated Amazon is inside a territory and should not be moved, except in situations of Zugzwang. Since this concept is way beyond the simplicity we search in the random games, we added the following rule to the random games:

– Any isolated Amazon should not be moved if possible

Obviously, if all Amazons are isolated, one has to be moved. But in this case, the game should be considered to be over: no player can now access to its opponent territory.

Tests and discussion of these two features will be discussed in section 4.

4 Experiments and Results

Due to the absence of popular Amazons servers and game protocol, testing of these improvements has been realised through self-play games against a standard version of our program. Each test consisted of runs of 250 games with 3 minutes per player, each player playing half of the time as first player. Some games were also played by hand against an other Amazons program: Invader [9], with an equivalent 30 sec/move time setting for both programs.

The standard version used for testing, later called Vanilla-Campya, uses a light version of the algorithm presented in section 2: Monte-Carlo without Tree-Search. Moves in the random games are split, and the evaluation of the games is given by their score. The results of the game-independent features are shown in table 1, and those of game-dependent features (liberty rule and pruning isolated Amazons moves) in table 2. Each feature or set of features was added to Vanilla-Campya to produce a new version of the program, and then tested against the Vanilla version.

Table 1. Results of Campya integrating some features against a standard MC version.

Feature tested	Win ratio
Not splitting moves in the random games	20,4%
Evaluation by Win-Loss ratio	43,6%
Evaluation by combining score and Win-Loss ratio	63,5%
Using tree-search	81,6%
Using tree-search and combined evaluation	89,2%
(*) Using tree-search, combined evaluation, and splitting moves in the tree-search	96%

The results obtained by the version not splitting moves in the random games are conform to our intuition, with only 20% of win against the Vanilla version.

Further tests (not included here) showed us also that, even with an equivalent number of playouts, the non-splitting version was behind. The results obtained using different evaluations are a bit more surprising: it seems that, for the game of Amazons, evaluation with a Win-Loss ratio is not the key, and that the score alone is not sufficient either. Finally, the results obtained by integrating Tree-Search are not surprising: a Tree-Search based version of CAMPYA is way above the others. Also, splitting the moves in the tree used by this version seems really effective, and not only because of the higher number of random games that CAMPYA could launch: as for splitting moves in the random games, even with an equivalent number of playouts, the non-splitting version was behind.

Table 2. Results of CAMPYA integrating some features against a standard MC+TS version.

Feature tested	Win ratio
Features (*) + liberty rule	94,4%
Features (*) + pruning	92,8%
Features (*) + pruning + liberty rule	96,4%

The version of CAMPYA using the features (*) in table 1 was used as a basis to test the knowledge-based improvements (liberty and pruning). Results obtained using them do not show a significant difference with the results given in table 1. However, considering that:

- Adding this form of external knowledge slowed the playouts, and thus did not permit us to launch as many as without, and
- The knowledge of liberties is especially useful against players who know how to exploit it, so not against VANILLA-CAMPYA,

we can still consider this integration of Amazons-knowledge in CAMPYA as being an improvement. Moreover, the first few games played against INVADER were terrific in the opening, because CAMPYA had no knowledge of Amazon imprisonment. Adding it permitted our program to perform better games against INVADER.

5 Integrating the Accessibility in CAMPYA

At this point, CAMPYA still lacked an important knowledge, used a lot in other programs. The accessibility to a square by a player can be defined as the minimum number of Amazon move a player has to perform to place an Amazon on this square. This number is set to infinite if the player has no access to the square. It is used by many programs, such as Amazong [8]

Accessibility is territory-related: if a player has a better accessibility to a square than his opponent, this square has a higher chance to be part of this player's territory at the end of the game than to be part of his opponent's. This goes even more true as the game reaches its end. Lacking this knowledge, our program could not understand the idea of potential territory, and thus was mostly crushed in the opening and middle game by other programs or good human players.

Integrating this knowledge in a Monte-Carlo architecture cannot be done easily: it requires lots of computations, and thus slows down the speed of the random games too much to be useful. However, it can be integrated as a new evaluation, which led us to this:

- random games are not any more evaluated by their a combination score + win/loss at the end of the game, but by the estimated score of the game after a fixed number of plies

Tests of this new feature have been done the same way as presented in section 4, with the difference that the reference version was not VANILLA-CAMPYA any more, but the version (1) of table 1. Results are shown in table 3.

Table 3. Results of CAMPYA integrating accessibility-based evaluation against its previous version, in number of games (average score).

Version (1)	Version (1) + accessibility evaluation
45 (3)	205 (12)

Using the accessibility heuristic as a score evaluator allowed CAMPYA to perform much better results, having more than 80% of win against its previous best version and losses by only a few points, any feature similar except for the evaluation. The games played against INVADER also showed us that its level increased and that it was now able to understand the concept of potential territory, still not being able, in its actual version, to perform better than INVADER, but showing lots of potential.

6 Conclusion and Future Works

We presented in this paper how to best integrate the Monte-Carlo method for the purpose of obtaining a good Amazons playing program. We discussed the main features, and proposed forcing moves by the liberty rule and pruning useless moves as ways to improve the level of the play. Finally, we proposed combining MC and an evaluation function as being the best way to obtain a good level program, thus using MC to explore a game tree and not any more to create an evaluation function.

Further works mostly include finding other light forms of knowledge to improve the random plays, specifically related to the endgame and the opening. Also, we would like to find the best way to combine Monte-Carlo tree-search and en evaluation function for the game of Amazons.

Acknowledgements

The picture of the Amazons boards in section 2 comes from the website www.solitairelaboratory.com

References

1. Bruce Abramson (1993), Expected-outcome: a general model of static evaluation, IEEE transactions on pattern analysis and machine intelligence 12:22, 182-193.
2. Henry Avetisyan, Richard J. Lorentz (2002), Selective search in an Amazons program, Computers and Games 2002: 123-141.
3. Bruno Bouzy (2005), Associating knowledge and Monte Carlo approaches within a go program, Information Sciences, 175(4):247257, November 2005.
4. Bruno Bouzy (2005), Move Pruning Techniques for Monte-Carlo Go, 11th Advances in Computer Game conference, Taipei 2005.
5. Remi Coulom (2006), Efficient Selectivity and Backup operators in Monte-Carlo Tree-Search, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy, 2006.
6. Sylvain Gelly, Yizao Wang, Remi Munos, Olivier Teytaud, Modifications of UCT with Patterns in Monte-Carlo Go, Technical Report 6062, INRIA
7. Levente Kocsis, Csaba Szepesvari (2006), Bandit based Monte-Carlo planning, 5th European Conference on Machine Learning (ECML), Pisa, Italy, Pages 282-293, September 2006.
8. Jens Lieberum (2005), An evaluation function for the game of Amazons Theoretical computer science 349, 230-244, Elsevier, 2005.
9. Richard J. Lorentz, INVADER, http://www.csun.edu/ lorentz/amazon.htm
10. Martin Muller, Theodore Tegos (2001), Experiments in Computer Amazons, in R.J. Nowakowski editor, More games of No Chance, Cambridge University Press, 2001, 243-260

Gaming Tools

Extended General Gaming Model

Michel Quenault and Tristan Cazenave

LIASD
Dept. Informatique
Université Paris 8, 93526, Saint-Denis, France
miq75@free.fr, cazenave@ai.univ-paris8.fr

Abstract. General Gaming is a field of research on systems that can manage various game descriptions and play them effectively. These systems, unlike specialized ones, must deal with various kinds of games and cannot rely on any kind of game algorithms designed upstream, such as DEEPER BLUE ones. This field is currently mainly restricted to complete-information board games. Given the lack of more various games in the field of General Gaming, we propose in this article a very open functional model and its tested implementation. This model is able to bring together on the same engine both card games and board games, both complete- and incomplete-information games, both deterministic and chance games.

1 Introduction

Many people are playing strategy games like Chess all around the world. Beyond the amusement they provide and their abstract forms, they surely lead to create some kind of reflection process that would be useful in everyday life. They build this way some kind of 'intelligence'. This is one of the main reason why they always are a prolific research field in computer science, which need highly efficient frameworks and especially in Artificial Intelligence which tries to define new ones.

Many Artificial Intelligence projects such as DEEPER BLUE are focused on specific games. This implies they use highly specific players, which use specialized data corresponding to some particular attributes of the game they are playing. Such players would certainly be efficient when confronted to their game, even to very similar ones, but they could not afford playing any other simply different games. However, since quite some time [1] a branch of Artificial Intelligence focuses on this problem and tries to develop players that would be able to play any kind of strategy games: General Gaming.

In General Gaming, as in the Metagame project [2], the objective is to compute players that would be efficient in various kinds of games. As we have seen, these players are not be implemented focusing on a specified game. This implies to define games engines which are able to deal with such players and to link them to the desired games. General Gaming's objective is to afford playing a maximum of different games, but card games, for example, are so much different from strategy games that nowadays, no general gaming project integrates them with strategy games. General Gaming engines focus only on strategy games, which is already a large and interesting field.

General game engines already exist. The General Game Playing Project from the Stanford University Computer Science Department [3] is a good example. Using games rules defined as logic programs, it makes play worldwide computer players on various strategy games, allowing players to know the game rules and to adapt themselves to be more efficient on a specific rule. Some commercial engines exist too, such as ZILLIONS OF GAMES [4], which has a specific functional language to describe a game's rule. Using a proprietary engine, he creates the game's engine and allows human players to play against their own proprietary computer player. A huge collection of games is already downloadable and is extended day by day by the users. However, these two projects have some limitations and use structures that restrict their usage to strategy games. Some of these restrictions are: no incomplete-information, no or very few chance and, for the second one, no way to create efficient connection games like Hex.

Based on this observation we have designed and implemented an engine based on three objectives. The principal one is to allow players to confront each other on many game kinds, including strategy games but also card games and board games such as Monopoly. Our second point is to give to players of various kinds the ability to simulate sequences of moves. The last but not the least point is to make such games and players easy to define. Such an engine is useful to compare players' efficiency on various kinds of games.

However, allowing players to confront each other on many games has some limits. We planned to integrate as many board games as possible, but computers are limited. We intend to manage finite strategy games such as Chess, card games, board games such as Monopoly and domino games. For different reasons, we limit our games to non-continuous boards, no query/answers games, no speech or dispute games, and no ultra-rich game universes as for role-playing games. Some games like wargames or deck collection games (such as Magic the Gathering) are left aside in the current version, but our structure intends to be easily adaptable to them.

This article presents the general gaming model we have defined. First the way we define games, then the way we connect players to games and eventually the main engine process. We conclude with a short discussion.

2 Games Descriptions

Any game is composed by exactly two things: A set of **Equipments**, with specific properties (boards, pieces, cards, dices) and the **Rule** of the game, which describes how players interact with components. Equipments which may seem to be very different can often be unified together. Here we present an organization of all the components we need to use to play a game, then we define what are game rules and eventually we present a simple rule as it is written in our engine.

2.1 Equipments

A strategy game often uses a board and some pieces. The board is an **Area** composed of a list of independent positions where the pieces can take place. These **Positions** are themselves relatives, they form a graph where nodes are positions and arcs are **Directions**. The pieces are **Elements** with different specifications. They are organized into **Assortments** of pieces, sometimes with many identical pieces, and sometimes no two pieces are the same. The Equipments we use here are: Areas, Positions, Directions, Elements, and Assortments. Areas are graphs of Positions and Directions, Assortments are composed of Elements.

A card game often uses a table with some defined placements and cards. It also uses some abstract Equipments such as players' hands which are sets of cards. The table is a kind of Area, generally but not systematically without Directions defined. The cards are Elements, organized into Assortments too. The abstract sets of cards are represented on tables Areas as Positions because it makes easier the representation of the game and the human interface. This implies that Positions can be occupied with multiple Elements, but this is already the case in some strategy games, so it implies no modification in our Equipments list. Moreover, we use exactly the same Equipments to define both strategy and card games.

However, some other Equipments are necessary to completely cover the field of board games: **Dices** as chance generators and **Score** markers to register score. At last, we need a **Player** which represents one of the active player in our system and a **Table** which is the container of all other Equipments. With all these Equipments, we could define almost all computable board games, if we specify some restrictions. Here is a recapitulated view of these Equipments with few spotted remarks:

1. Area: A picture associated with a Position graph.
   – Area, Direction and Positions define the board or the card table.
   – Almost all games have at least one Area, but this number is totally free.
2. Position: Node of Area's graph.
3. Direction: Oriented and labeled arc of Area's graph.
   – Positions could be empty, occupied with one Element, or occupied with an ordered list of Elements.
4. Assortment: A list of Elements used in the game. This could represent a card deck, or a stock of go stones.
   – Almost all games have at least one Assortment, but this number is totally free.
5. Element: These are cards, pieces, stones, etc.
   – Elements must be able to receive any kind of attribute, with any possible value. This allows to define cards (with colors, values, ranks, etc.) or Chess pieces (with type, cost, etc.). Actually, this restriction is extended to all the Equipments in the game, for easiness in the rule definition process.
6. Dice: A specific equipment to generate chance.
7. Score: A specific equipment to keep score data as in most card games.
8. Player: One more specific equipment representing a player in the game.
9. Table: This last equipment is a container of all other equipments in play.

All of these Equipments possess methods that return other related Equipments. This way the engine and the Rule can navigate through them ad libitum.

2.2 Rule

The second thing defining a game is **Rule**. First it defines the number of players. Then it defines a graph of successions of player's turns.[1] Nodes of this graph are play stages

[1] Many strategy two players games simply alternate the two players roles, but some complex games like traditional Mah-Jong needs all the potential of graphs.

where one or more players may choose to execute Actions. Arcs are brace of players and possible Actions.

Then the rule defines the initial state of the Equipments[2] and final conditions with associated winners. These parts of the rules need the use of a method call. The last thing defined by the rule is the explanation of legal Actions. These legal Actions are the link between initial state and final states described with final conditions. Here again, the use of a method call to create the list of legal Actions is coerced.

Method calls are needed to define initial states, Actions and final conditions. These methods must be defined in the rules objects and will always have as single argument a Table. This argument refers to all Equipments defined in the Rule and used in the play. The method must return new built objects corresponding to atomic **Actions** that alter the Table and correspond to players moves. These Actions could be any ordered combination of any number of Actions to ensure complex Actions ability to be defined. The initial state method must return exactly one Action. The final condition method return nothing or one special end game Action.The moves methods must return the list of actual legal Actions for the current stage of the play.

The possible Actions and their effects are:

1. Pass: Nothing to do.
2. Move: Move any number of Elements from a Position or an Assortment to another.
3. Score: Mark points in a Score Equipment.
4. FinishPlay: Declare some winners, some losers or a draw game.
5. Set: Add an attribute to any Equipment.
6. Del: Removes an attribute to any Equipment. Access to these attributes is ensured by methods in Equipments. Here are just defined Actions that alter the Equipments.
7. Distribute: Distributes all Elements from a Position or an Assortment to a list of Positions or Assortments and affects them a new owner relative to the Positions.
8. Sort: Sort the list of Elements in a Position or an Assortment.
9. Shuffle: Shuffle the list of Elements in a Position or an Assortment.
10. Roll: Randomly changes the value of a Dice list.

2.3 Example

Algorithm 1 is an example of the full definition file for a rule. The game is basic Tic-tac-toe. The language used is python. The `board` and `turns` values respectively describe the board graph and the turn order arc. Inline tools are provided to easily generate these lists but the use of such lists ensures that any board or turn order graph can be defined even when automatic method fails.

The `defineEquipments` method selects the Equipments used in the game. The Assortment last argument is the Elements layout. The two last methods define the final conditions and the legal Actions.

Notice the way the play data are accessed: `table.getPositions()`. Such as in `table.getElement(player=table.getCurrentPlayer())`, Equipments methods may have arguments to restrict returned Equipments on any attribute values.

[2] Equipments are indeed first defined in the rule too, so the rule is enough to fully define a game.

This short page is enough to create a complete game with our engine. The complex parts of code of Algorithm 1 are detailed in Appendix A.

```
from rule import *

board=[ ('A1', (60, 60), [('H', 'B1'), ('V', 'A2'), ('B', 'B2')]),
        ('B1', (150, 60), [('H', 'C1'), ('V', 'B2'), ('B', 'C2')]),
        ('C1', (240, 60), [('V', 'C2')]),
        ...]
turns=[ ('wait Cross', True, True, [('Cross', TicTacToe.move, 'wait Circle')]),
        ('wait Circle', True, True, [('Circle', TicTacToe.move, 'wait Cross')]) ]
pawns=[ ('X', 'Cross', 'images/cross.gif'),
         ('O', 'Circle', 'images/circle.gif')]

class TicTacToe (Rule):
    def __init__(self):
        self.name = 'Tic Tac Toe'
        self.players = ['Cross', 'Circle']
        self.turns = turns

    def defineEquipments(self, table):
        table.addEquipment(Area('images/ttt_board.gif', board))
        table.addEquipment(Assortment(pawns, ['name', 'player', 'image']))

    def playResult(self, table):
        action=table.getLastPlayerAction()
        if action!=None and table.hasNewLine([move.getPositionTo()], 3,
                      elementRestraints={'player': table.getCurrentPlayer()}):
            return FinishPlay(table, table.getCurrentPlayer())

    def move(self, table):
        res=[ ]
        for pos in table.getPositions():
            if pos.isEmpty():
                res.append(Move(table.getAssortment(), pos,
                        table.getElement(player=table.getCurrentPlayer())))
        return res
```

ALG. 1: Tic-Tac-Toe Class.

3 Players Descriptions

Here is a quick list of possible players we have started to develop and integrate in our general gaming model. All these methods can be easily combined:

1. Alpha-Beta, Min-Max, tree exploration based,

2. Monte-Carlo methods,
3. Neural Networks,
4. Genetic Algorithms and Genetic Programming.

Our model is based on functional rule description and step by step play unfolding. At any time in the game when some player can make an Action, this player is called with a list of each player possible Actions computed following the rule definition on the players variant of the Table. The player has to send back the Action he prefers. He may also launch a simulation of an Action and his consequences on the play. He could pursue this process anytime he wants and explore the play tree. For incomplete-information games the player sees all unknown information trough a randomly generated possible Table state. This unknown part of the play can be shuffled anytime to simulate another possible situation.

As for games, there are some limits to our application and the player that we could connect to it. One is that our model is based on functional rule description and step by step play deployment. This implies that we do not provide tools for analyzing game rules before the play begins. Actually no access to these data is provided yet and it can be easily improved, but it is complex enough to implement this model without querying about pre rule analyzer players. The other point is that our players are highly integrated in our game engine and that the engine is in charge of generating possible Actions for the player, even in simulations. Detaching players to try to solve this problem is one of the planned evolution of our engine.

Our players are connected to our engine with some few methods:

1. `doAction`: Play the selected player's Action.
2. `doSimulateAction`: Play any legal Action and compute the next possible Actions for all players, modify only the players specific Table.
3. `undoSimulateAction`: Undo the last simulated Action and restore specific Table and next possible Actions.
4. `getChoices`: Return the list of all possible players Actions corresponding to the current simulation or play.
5. `getEval`: Read the game engine's result on current simulation or play.

4 Engine Description

In this section we will focus on our game engine. First we will explain its global behavior, then we will present how we have implemented it and how we want to use and improve it later.

4.1 Main Loop of the Engine

The Algorithm 2 presents the main tasks of the Engine. After having initialized the rule object, the engine uses its attributes to define the different parts of the play: the turn order graph and the Tables related Equipments. The turn order graph leads the main course of the events by defining the possible players and the possible Actions during each play step. In order to do that, the engine needs to apply the rules Action creation method on each player's Tables (lines 7 and 8).

Then the engine calls the players to let them define the Action they want to realize (lines 9 and 10). During this phase, each player can use its own Table to manage Action simulations. Then the engine deals with different priority kind of rules to select the next legal Action in the players answers (Line 11). There are two possible ways to select the Action, one is to choose the faster player (this allows to compare quick-thinking players to deep thought ones on fast based games). The other way is to describe priority rules in the rule file as for in traditional Mah-Jong.

In incomplete-information games the player has in his Table one possible distribution of the Elements he does not know. During the creation of Actions (relatives to player's Tables but equivalents to the engine selected one)(line 12) there is a coherency engine which modifies any player's Table[3] so that these Tables correspond to the desired Action.

Then, the program loops until the engine detects a FinishPlay Action returned by `Rule.playResult()`(line 6).

```
1   Create $rule$ using $rule.\_\_init\_\_()$
2   Create $turn\_graph$ using $rule.turns$ and select start node
3   Create $Table[engine]$ using $rule.defineEquipments$
4   For each $player$ in $rule.players$:
5       Create $Table[player]$ using $rule.defineEquipments$
6   While $rule.playResult(Table[engine]) == None$:
7       For each $arc$ in $turn\_graph.current\_node$:
8           Create $possible\_actions[player]$ using $(Table[arc.player], arc.method)$
9       For each $player$ having $possible\_actions$:
10          Select $favorite\_action[player]$ using $Table[player]$
11      Select one player's $favorite\_action$
12      Recreate $selected\_favorite\_action$ on all $Tables$
13      Apply $selected\_favorite\_action$ on all $Tables$
14      Update $turn\_graph.current\_node$
```

ALG. 2: Engine main loop.

4.2 Implementation of the Engine

At the moment, the engine implementation is in progress in the Python language. Three games are defined: A Tic-Tac-Toe which is described by Algorithm 1, a Go-Moku which uses a very similar file and a Chinese-Poker which is a four players card game using poker combinations. One player is implemented too: A Min-Max player with op-

[3] e.g. Before George plays a 3♠, Jane was thinking he had in his hands only a 2♢ and a 7♡. When George play his 3♠, Jane's knowledge on Georges hands would change this way: 3♠ anywhere guessed position would be swapped with either the 2♢ or the 7♡ ones in Jane's idea of George's hands. Then George could play this card.

tional Alpha-Beta cut. A Monte-Carlo player is on the point of being added, as soon as multiple Table control is fully realized.[4]

All Equipments are already defined and created, except the Dice. All Equipments are linked to some others ones. The state of these relations represents the state of the table during the play. Rule can use many Equipments methods to test this state. For instance, these are few of the Position Equipment methods to illustrate the principle:

1. `getArea()`: Returns the Area which the Position depends on.
2. `getElements(restraints)`: Returns the list of Elements played on the Position. Restraint is an optional dictionary of attributes which filters the returned Elements.
3. `getDirections(name, orientation)`: Returns the possible Directions that links this Position with others on its Area.
4. `getOppositeDirections(direction)`: Returns the opposite Directions (if any) to the one given as argument.

All Actions are already defined too. They have two main methods allowing the engine to really alter Tables. One is `doAction()` which performs the desired action and the other is `undoAction()` which restores the Table in its previous state. This way the engine manages players simulations. Actions have the ability to add themselves to each others, so `Move()+Set()` is seen by the engine as only one Action. There are many other Actions methods used by the engine (to manage graphic interface for instance) we choose not to describe here.

The engine uses a few more classes to implement the model: Graphic interface, engine which manages the main loop, and graphs are used too. Some other no fundamental tools are provided: It is possible to define options in game rules (such as exactly or at least five pawns on Go-Moku). These options are defined in the `rule.__init__()` method and must be chosen before the engine starts to play. A tool is provided to automatically generate the Area's graph definition lists, based on dimensions. Some complex non required methods are provided (`Table.hasNewLine()` for instance) to make easier rule creation.

Despite of its early development stage, this engine can already manage both complete-information strategy games such as Chess and both incomplete-information card chance based games such as traditional Mah-Jong. Today, as far as we knew, there is no other engine with similar proficiency.

4.3 The Future of the Engine

There are many possible uses or upgrades to this engine. We present now the main ones.

The first use is for Artificial Intelligence benchmarking. With the capability to create very different kinds of games, based on opposite moves process or goals and the capability of develop various General Gaming players, this engine would be useful to compare their efficiency, their robustness and even their creativeness facing various

[4] Each player has its own Table, which corresponds to what he knows of the play. These Tables are highly cross-referenced to each others to manage incomplete-information games coherence engine. This part is in debugging stage.

problems which have often already been classified [5]. The model was first developed in this perspective.

Another evident possible use is for entertainment of many players around the world, connected to many possible games against efficient computer engines. The easiness in the game's rule creation would probably lead to such an amazing collection of games than for ZILLIONS OF GAMES if we bother to distribute this engine as they did.

Furthermore, this engine is quite young and it would be instructive to develop it in some new ways, in extending the range of possible games definition, or in players interface. Some evolutions concerning games could be the integration of collecting games, such as wargames or card collecting games, where the players must first define their army or their deck following specific rules before confronting other players with their owns. Another game interface evolution could be the management of continuous boards with some geometric tools in place of Areas and their Positions lists.

For the engine upgrading, it would be pleasant to tear apart the players and the engine, in order to allow engines players to perform theirs owns play explorations. This would lead to a more open engine players system, with capability of pre rule analysis.

There are many much more ways to improve this system and not enough room here to describe them all.

5 Discussions

Before concluding this article, we suggest some short discussion about our engine in the form of short queries with their answers.

– Is this model better than Stanford ones?
- No, it is not better, it is different. Stanford general game playing model uses rules in logic form, is limited to simple strategy games, and allows players to analyze rules. Our model is driven by the intention of playing easily almost any game and our players are restricted to choose the best Action in a possible tree of Actions. The two model are completing each other.

– Isn't the model too heavy?
- No, the model isn't heavy. It's a relatively light model to cover the large field of possible games. However, as the engine must compute all players simulations trees, it is quite long to play a game. This is one of the reasons why one of the next upgrade would probably be the full players parting from the engine.

– Is it really interesting to test Monte-Carlo methods on complete-information games or Alpha-Beta methods on chance incomplete-information games?
- Yes, good results have been obtained on go with Monte-Carlo players [6].

6 Conclusion

Nowadays, computers are very effective in most board games. The issue of years of research in such fields as Computer Science and Artificial Intelligence is that computers are capable to defeat the best humans players in almost all games (with some exceptions nevertheless). This shows how the advancement of theses sciences are awesome. But all these engines are specifics to their games, and do not reflect even a part of the human mind which is able to be (almost) good in any game, with the same mind.

So, it is the next step to explore the huge field of general solving methods as general gaming tries to address. We have done one more step by creating, implementing and testing a new model which is the first to allow the use of such various games as strategy, card and board games. Furthermore, we have opened the way for collecting games.

This way only, Computer Science and Artificial Intelligence will continue on their march to maybe beat, one day, human mind not because they are faster and more robust systems, but because they are more malleable and adaptive ones.

A Algorithm 1 Code Explication

Some complex calls in Algorithm 1 are detailed here:

- Line 11: `Area` is an Equipment. The arguments are the board picture and a list of positions data. The position layout is (name, coordinate, list of (direction name, direction goal)).
- Line 12: `Assortment` is an Equipment. The arguments are a list of pieces data and the corresponding layout. Some attributes (such as image) must been defined in the layout.
- Line 14: `Table.getLastPlayerAction()` returns the previous move in the game. This test is needed to control that we are not before the first move.
- Line 15: `Table.hasNewLine()` returns a boolean defining if a line is detected into an Area. Arguments are the list of positions that may be in the line,[5] the size of the line, and some restraints which must be checked for one Element at each Position on the line. Here the restraint is the name of the player that owns the Element.
- Line 16: `FinishPlay` is an Action which defines the winner, which here is the current player.
- Line 21: `Move` is an Action. Arguments are the source, the target and the Elements moved. Here, we move one Element (returned by `table.getElement()`) from Table's Assortment to the Table's search current Position (line 19).

References

1. Pitrat, J.: Realization of a general game-playing program. In: IFIP Congress (2). (1968) 1570–1574
2. Pell, B.: A strategic metagame player for general chesslike games. In: AAAI. (1994) 1378–1385
3. Genesereth, M.R., Love, N., Pell, B.: General game playing: Overview of the aaai competition. AI Magazine **26**(2) (2005) 62–72
4. Lefler, M., Mallett, J.: Zillions of games. Commercial website http://www.zillions-of-games.com/index.html.
5. Boutin, M.: Le Livre des Jeux de pions. Livres de jeux. Éditions Bornemann (april 1999)
6. Bouzy, B., Helmstetter, B.: Monte-carlo go developments. In van den Herik, H.J., Iida, H., Heinz, E.A., eds.: ACG. Volume 263 of IFIP., Kluwer (2003) 159–174

[5] Typically the last played Positions, to avoid useless searches on all Area's Positions.

GTQL: A Query Language for Game Trees

Yngvi Björnsson and Jónheidur Ísleifsdóttir

Department of Computer Science, Reykjavik University
{yngvi,jonheiduri02}@ru.is

Abstract. The search engines of high-performance game-playing programs are getting increasingly complicated as more and more enhancements get added. To maintain and further enhance such engines is an involved task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. In this paper we introduce GTQL, a query language specifically designed to query game trees. A software query tool based on GTQL both helps program developers to gain added insight into the search process, and makes regression testing easier. Experiments using the tool to query game-tree log files generated by the public-domain chess program Fruit, show GTQL to be both expressive and efficient in processing large game trees.

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 programs, become quite sophisticated when all the necessary bells and whistles have been added. To maintain and further enhance such complicated software is an involved task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. For example, different search enhancements affect each other in various ways, and changing one may decrease the effectiveness of another. Similarly, tuning an evaluation function to better evaluate specific types of game positions may have adverse side effects on others.

A standard software-engineering approach for verifying that new modifications do not break existing code is to use *regression testing.* To a large extend this approach is what game-playing program developers use. They keep around large suits of test positions and make sure the modified programs evaluate them correctly and that the search finds the correct moves. 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 and they can stay hidden for a long time without surfacing. These 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 instead seriously degrade the effectiveness of the search. To detect such anomalies one typically must explore and/or gather statistics about the search process.

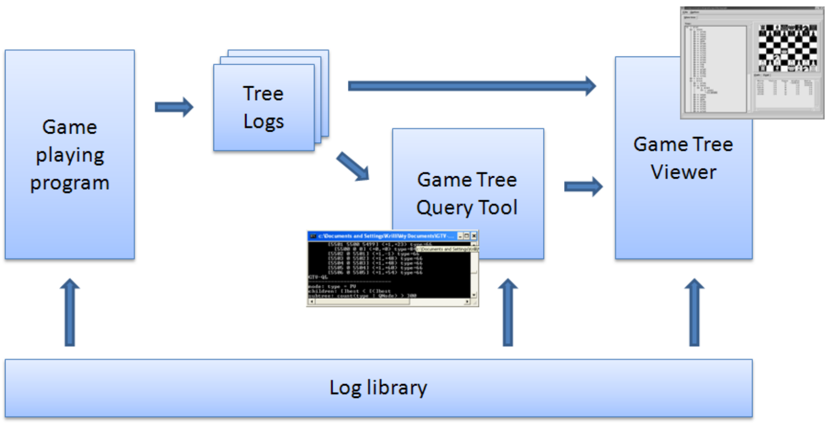


Fig. 1. The GT-tool suite.

In this paper we introduce *Game-Tree Query Language (GTQL)*, a language for querying game-tree log files. This language is a part of a larger suit of tools intended to alleviate the difficulty of debugging large game trees. The query language allows the game-program developers to gain better insight into the behavior of the search process and makes regression testing easier. A programmer can keep around a set of pre-defined queries that check for various 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 with search trees, and the queries are then run against the logs to verify that the search is behaving in accordance to expectations.

The paper is organized as follows. In the next section we give a brief overview of the larger suit of tools GTQL belongs to, then we discuss the syntax and semantics of GTQL and evaluate the performance of our implementation. Finally, we briefly survey related work before concluding.

2 GT-Tools Overview

GTQL is an integral part of a larger collection of tools for aiding researchers in the analysis and visualization of large game trees. An overview of this suit of tools is shown in Figure 1. It consists of a library for logging game-tree information, the game-tree query tool for processing GQTL queries, and a game-tree viewer for graphically viewing game-tree log files and query results. The game-playing program's developer adds to its program a handful of calls to the log library using a well-defined API (the API details are outside the scope of this paper, but an example of its use is provided in an appendix). The game-playing program then generates files containing the search trees, which can then be either viewed with the game-tree viewer or analyzed using the game-tree query tool.

3 Description of GTQL

In this section we first describe the syntax and the semantics of GTQL. Then, to highlight the expressiveness of the language, we give several examples of GTQL queries that one might want to ask about game trees generated by alpha-beta based search engines. A complete EBNF description of the GTQL syntax is provided in an appendix.

3.1 Syntax and Semantics

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 — if the expression is empty then the keyword (along with the following colon) may be omitted. Valid expressions must be formed such that they evaluate to either true or false. The language is case sensitive.

A query is performed on a game-tree file. The corresponding game tree is traversed in a left-to-right depth-first manner and the node-expression part of the query is evaluated for each node in a pre-order fashion (i.e. on the way down). If the node-expression evaluates to true, then the child and subtree parts of the query are evaluated as well (we use a one-pass algorithm for this as described in the next section). A node fulfills the query if all expression parts evaluate to true for the node. The query returns either a set of nodes that fulfill the query or, in case the node-expression is an aggregate expression, the number of nodes that fulfill the query.

Expressions consist of *attributes*, *constants*, *operators*, and *functions*. *Attributes* refer to the attribute values of the nodes stored in the file being queried. For each node several attributes are stored, two of which are always present (*node_id* and *last_move*) and others that are optional. The optional attributes are typically algorithm/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.). The name of the attributes must 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 that refer to numeric integral types. The same naming convention is used for constant names as for attribute names. Information about attribute and constant names available in a query are stored in the game-tree file being queried.

Table 1. Operators listed by precedence.

Operator	Type	Arity
[ ], [<]	Hierarchical	unary
|	Attribute	binary
<,>, >=, <=, =, ! =	Relational	unary
not	Logical	unary
and	Logical	binary
or	Logical	binary

In the current version of the language attribute values, like constants, can only be numeric integral types.

The language operators fall into four categories: *attribute*, *hierarchical*, *relational*, and *logical* operators. They are listed in Table 1 in a decreasing order of precedence. The evaluation of operators of equal precedence is right-to-left 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, [], stands for the child node of the current node that is being evaluated. For example, the child expression `count([]type=type)` counts the number of children that are of the same type as the current node (in child-expressions, attributes without a prefix refer to the current node). The second operator, [<], stands for the previously evaluated child. The sub-expression `([<]type=[]type)` thus asks about two consecutive child nodes of the same type. The *attribute* operator | is essentially an inclusive bitwise or, and is used to extract flag bits out of attribute fields. For example, a single node may be flagged as being simultainiously a pv-node and a quiescence node. The *relational* operators test for equality or inequality of attributes, constants, and sub-expressions, and the *logical* operators allows us to form combined Boolean expressions. Parentheses can be used to control precedence and order of evaluation.

There is only one function in the language, the `count(<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 with 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 to this is when the function is used in a node-expression (e.g. `node: count(type|typePVNode)`). In that case, the query returns the number of nodes fulfilling it. Node-expressions can be either aggregate or regular, whereas child and subtree expressions must contain an aggregate function to be meaningful. The word *count* is a reserved keyword in the language.

3.2 Example Queries

In the examples below we assume that the search is logging for each node information about its ply-depth in the search tree (*depth*) and flags indicating the node type (*type*). A node can be simultaneously flagged as being of several types, e.g. a pv-node and a quiescence node.

Query 1 In this first example query we want to find whether the game-playing program is extending the search too aggressively. We form the query by asking for nodes where the *depth* attribute has a value greater than some deep threshold value, excluding nodes in the quiescence search.

```
node: depth>=10 and not type|typeQuiescenceNode
```

Query 2 As in the previous query, we are interested in identifying subtrees where too aggressive expansion occurs. However, now we want to identify places where the quiescence search is too aggressive. We form the query by asking for quiescence root nodes having a subtree larger than 50 nodes.

```
node: type|typeRootQuiescenceNode
subtree: count(*) > 50
```

Query 3 In this example, we want to identify nodes where the principal variation of the search changes frequently. Note that the node-expression part is not necessary for retrieving the answer, however, it is beneficial to include it as it constrains the search space of the query such that the child-expression is evaluated only at pv-nodes.

```
node: type|typePVNode
child: count( []type|typePVNode ) > 2
```

4 Implementation of the Query Tool

In here we describe the implementation of the game-tree query tool. The tool is written in C++ and its task is twofold: first to parse the query and construct a parse tree, and second to traverse the game-tree file and identify nodes where the parse-tree evaluates to true. We first describe the building of the parse tree, and then the game-tree traversal and evaluation.

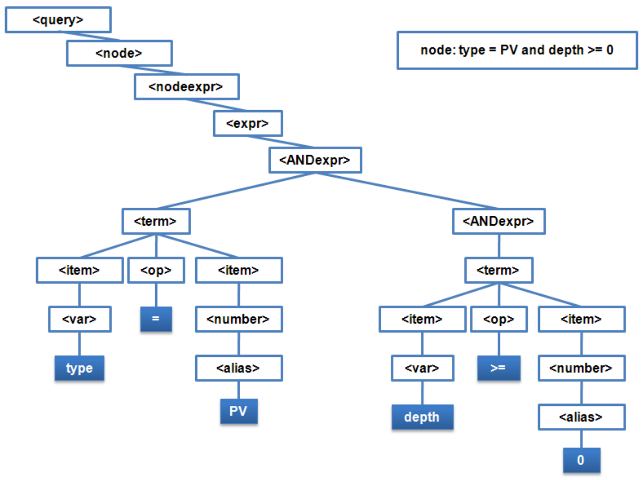


Fig. 2. Example parse tree.

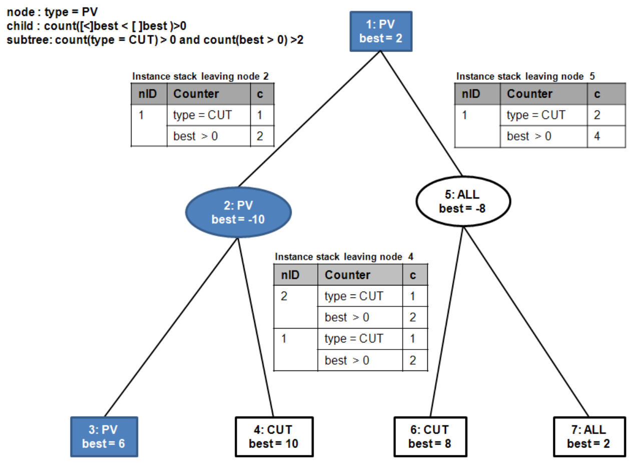
4.1 Parsing a Query

A recursive-decent parser is used to parse the query and build a parse tree. The parse tree consists of several different types of parse nodes. An example tree is shown in Figure 2, along with the query being parsed. Given a set of node attribute values, a parse-tree expression evaluates to either true or false. For example, in the above example the attribute value of both the *type* and *depth* fields are required for evaluating the query. A special provision must be taken for queries containing the aggregate function *count*; in that case, in addition to the attribute values, a special structure containing count information accumulated in the subtrees must be provided.

4.2 Evaluating a Query

The parse-tree is evaluated for all nodes in the game tree. The algorithm we use allows us to do that in a single depth-first left-to-right traversal of the game tree. As the algorithm traverses down a branch, the node-expression part of the query is first evaluated. If it evaluates to true, and the query contains either a child or a subtree expression part, an instance of that query is kept active while the node's subtree is traversed. This is because the child and expression part can only be evaluated after the subtree has been traversed, using information accumulated during the traversal. This means that multiple query instances can be active at the same time. The number of simultaneously active instances is though bounded by the maximum game-tree depth. A stack is used to keep track of active query instances.

An example of a tree traversal and parse-tree evaluation is given in Figure 3. The node-expression part of the query looks for pv-nodes. The root is a pv-node



Instance stack leaving node 2

nID	Counter	c
1	type = CUT	1
	best > 0	2

Instance stack leaving node 5

nID	Counter	c
1	type = CUT	2
	best > 0	4

Instance stack leaving node 4

nID	Counter	c
2	type = CUT	1
	best > 0	2
1	type = CUT	1
	best > 0	2

Fig. 3. Traversing the data tree using instances.

so a new query instance is created on the stack. An instance contains an entry for each count function appearing in the query's child- and subtree-expressions. The entry stores a pointer to the count subexpression (more specifically, to the parse-node representing the root of the subexpression), and a counter variable initialized to zero (the *c* field in the figure). The pointer allows us to evaluate the count subexpressions for all nodes in the corresponding subtree, and the counter variable accumulates information about for how many nodes in the subtree the subexpressions evaluate to true.

The traversal continues down the left branch, and as the node-expression is also true for node 2 and 3, instances are created on the stack for those nodes as well. Now, because a leaf has been reached, the tree traversal algorithm starts to backtrack. However, before backtracking it pops the top query instance off the stack if the instance belongs to the current node, and finishes evaluating that query. Also, all remaining subexpression count instances on the stack are updated. 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.

4.3 Performance Evaluation

We have experimented with the query tool on actual game trees generated by the public-domain chess program Fruit (version 2.1) [7]. On large game trees, containing tens of millions of nodes, the processing time is still acceptable; the tool processes approximately 100,000 nodes per second even when evaluating complex queries on a middle-end PC desktop computer (2.0GHz CPU). The memory footprint of the tool is also low because of sequential processing. Also of interest is that we have been able to identify several positions where suspect behavior occurs, for example where the quiescence search explores exceedingly large trees.

5 Related Work

To the best of our knowledge GTQL is the first language specifically designed for querying game trees. However, there exists several query languages for tree structures. The best know is probably XPath [4], which was designed to query XML data. The XPath language has been extended to increase its expressiveness (XQuery [3]), to better handle sub-documents (XSQirrel [8]), and to use with linguistic data (LPath [1]). Although the above languages all provide query mechanism for referencing children, parents, descendants, and ancestors, then they do not allow aggregation. Also, they are primarily designed for use on relatively small and shallow trees, and consequently can afford complex expressions. GTQL is on the contrary designed for use on large trees, and the query expressiveness is tailored such that the queries can be evaluated in an one-pass left-to-right tree traversal.

In addition to query languages, game-tree visualization tools are helpful for gaining insight into the search process. There exists several such tools, some of which draw the trees in a graph-based hierarchical fashion [2, 6] as well as others that use more innovative 2D surface representations [5].

6 Conclusions

We introduced a new query language specifically designed to query game trees. Such a language is useful to both help program developers gain added insight into the search process, and in assisting with regression testing. Experiments using a software tool based on the language show it to be both efficient in processing large game trees, and to be expressive enough to detect many interesting search anomalies. We are still working on improving the processing performance of the query tool even further. The plan is to make the entire GT-Tool suite of tools available to the game-developers community, including the GTQL query tool.

7 Acknowledgment

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 number MIRG-CT-2005-017284.

References

1. 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, ACM (January 2005) 35–46
2. Björnsson, Y., Ísleifsdóttir, J.: Tools for debugging large game trees. In: Proceedings of The Eleventh Game Programming Workshop, Hakone, Japan (2006)
3. Chamberlin, D.: XQuery: An XML query language. IBM Systems Journal **41**(4) (2002) 597–615

4. Clark, J., DeRose, S.: XML path language (XPath) 1.0. Technical report, W3C Recommendation (1999)
5. Coulom, R.: Treemaps for search-tree visualization. In Uiterwijk, J.W.H.M., ed.: The Seventh Computer Olympiad Computer-Games Workshop Proceedings. (2002)
6. Fortuna, A. Internet Resource http://chessvortex.com/chant (2003) CHANT: A Tool to View Chess Game Trees.
7. Letouzey, F. Internet Resource http://www.fruitchess.com (2005) Fruit Chess.
8. Sahuguet, A., Alexe, B.: Sub-document queries over XML with XSQirrel. In: WWW '05: Proceedings of the 14th international conference on World Wide Web, New York, NY, USA, ACM Press (2005) 268–277

A Appendix

A.1 Logging Game Trees

This is an example to show how to use the game-tree log interface in a game program. We use here a simple iterative deepening and minimax procedure for demonstration purposes (and omit various details that are not relevant for our demonstration purposes). Essentially, one must create a handle in the beginning (and delete in the end), open a new file for each search iteration, and call special functions when entering/exiting a node. The user collects the information he or she wants to log in the structure *data*.

```
/*  Example TicTacToe program. */
#include "gt_log.h"
...
GTDataDescript gtDataDescr  =      /* <= GTL data description */
  { "TicTacToe", sizeof(Data_t), 0, {}, 5,
       { ...
         { "depth", offsetof(Data_t,depth), sizeof(int) },
         { "best" , offsetof(Data_t,best),  sizeof(int) },
         { "type" , offsetof(Data_t,type),  sizeof(int) } }
  };
GTLogHdl hGTL;   /* <=  Game-tree log handle. */

Value Minimax( Position_t *pos, int depth, Move_t move_last ) {
   Data_t  data;  /* <= GTL data record, user defined. */
   ...
   data.depth = depth;
   gtl_enterNode( hGTL, move_last );   /* <= GTL enter node.*/
   ...
   n = generateMoves(pos, moves);
   for ( i=0 ; i<n ; i++ ) {
      makeMove( pos, moves[i] );
      value = -Minimax( pos, depth-1, moves[i] );
      ...
      retractMove( pos, moves[i] );
   }
   ...
   data.best = best;
   gtl_exitNode( hGTL, &data );   /* <= GTL exit node */
   return best;
}

Value IterativeDeepening(  ... ) {
   ...
   for ( iter=1 ; iter<=maxIter ; ++iter ) {
      ...
      gtl_startTree( hGTL, filename, strFEN );  /* <= GTL new tree */
      ...
      value = Minimax( &pos, iter, NULL_MOVE );
      ...
      gtl_stopTree( hGTL );     /* <= GTL close tree */
   }
   ...
}

int main() {
 ...
 hGTL = gtl_newHdl( "TicTacToe", &gtDataDescr ); /*<= GTL new handle*/
 if ( hGTL == NULL ) exit(EXIT_FAILURE);
 ...
 gtl_deleteHdl( &hGTL );    /* <=  GTL delete handle */
 ...
}
```

A.2 EBNF for GTQL

```
<query>    :=    <node>
                 | <child>
                 | <subtree>
                 | <node><child>
                 | <node><subtree>
                 | <child><subtree>
                 | <node><child><subtree>

<node>     :=    'node' ':' <nodeexpr>

<child>    :=    'child' ':' <childexpr>
```

```
<subtree>          :=   ’subtree’ ’:’ <treeexpr>

<nodeexpr>         :=   <expr>
                        | ’count’ ’(’ <expr> ’)’

<expr>             :=   <ANDexpr> ’or’ <expr>
                        | <ANDexpr>
                        | <wildcard>

<ANDexpr>          :=   <term> ’and’ <ANDexpr>
                        | <term>

<term>             :=   <item> <op> <item>
                        | ’not’ <term>
                        | ’(’ <expr> ’)’

<item>             :=   <var>
                        | <number>

<treeexpr>         :=   <treeANDexpr> ’or’ <treeexpr>
                        | <treeANDexpr>

<treeANDexpr>      :=   <treeterm> ’and’ <treeANDexpr>
                        | <treeterm>

<treeterm>         :=   <treecountitem> <op> <treecountitem>
                        | ’not’ <treeterm>
                        | ( <treeexpr> )

<treecountitem>    :=   ’count’ ’(’ <expr> ’)’
                        | <number>

<childexpr>        :=   <childANDexpr> ’or’ <childexpr>
                        | <childANDexpr>

<childANDexpr>     :=   <childterm> ’and’ <childANDexpr>
                        | <childterm>

<childterm>        :=   <childcountitem > <op> <childcountitem>
                        | ’not’ <childterm>
                        | ’(’ <childexpr> ’)’

<childcountitem>   :=   ’count’ ’(’ <siblingexpr> ’)’
                        | <number>

<siblingexpr>      :=   <siblingANDexpr> ’or’ <siblingexpr >
                        | < siblingANDexpr >
                        | <wildcard>

<siblingANDexpr>   :=   <siblingterm> ’and’ <sibingANDexpr>
                        | <siblingterm>
```

```
<siblingterm>    :=    <siblingitem> <op> <siblingitem>
                       | 'not' <siblingterm>
                       | '(' <siblingexpr> ')'

<siblingitem>    :=    <sibling><var>
                       | <var>
                       | <number>

<number>         :=    - <digits>
                       | <digits>
                       | <alias>

<sibling>        :=    '[<]' | '[]'

<op>             :=    '=' | '!=' | '<' | '>' | '>=' | '<=' | '|'

<wildcard>       :=    '*'

<digits>         :=    [ '0'- '9' ] [ '0'- '9' ]*

<var>            :=    [ 'A'-'Z', 'a'-'z' ] [ 'A'-'Z', 'a'-'z', '_', '0'-'9' ]*

<alias>          :=    [ 'A'-'Z', 'a'-'z' ] [ 'A'-'Z', 'a'-'z', '_', '0'-'9' ]*
```

Video Games

Predicting Success in an Imperfect-Information Game

Sander Bakkes, Pieter Spronck, Jaap van den Herik, and Philip Kerbusch

Universiteit Maastricht / MICC-IKAT
P.O. Box 616, NL-6200 MD Maastricht, The Netherlands
{s.bakkes,p.spronck,herik}@micc.unimaas.nl, p.kerbusch@student.unimaas.nl

Abstract. One of the most challenging tasks when creating an adaptation mechanism is to transform domain knowledge into an evaluation function that adequately measures the quality of the generated solutions. The high complexity of modern video games makes the task to generate a suitable evaluation function for adaptive game AI even more difficult. Still, our aim is to fully automatically generate an evaluation function for adaptive game AI. This paper describes our approach, and discusses the experiments performed in the RTS game Spring. TD-learning is applied for establishing a unit-based evaluation term. In addition, we define a term that evaluates tactical positions. From our results we may conclude that an evaluation function based on the defined terms is able to predict the outcome of a Spring game reasonably well. That is, for a unit-based evaluation the evaluation function is correct in about 76% of all games played, and when evaluating tactical positions it is correct in about 97% of all games played. A straightforward combination of the two terms did not produce improved results.

1 Introduction

Modern video games present a complex and realistic environment in which game AI is expected to behave realistically (i.e., 'human-like'). One feature of human-like behaviour of game AI, namely the ability to adapt adequately to changing circumstances, has been explored with some success in previous research [3, 4, 6]. This is called 'adaptive game AI'. When implementing adaptive game AI, arguably the most important factor is the evaluation function that rates the quality of newly generated game AI. An erroneous or suboptimal evaluation function will slow down the learning process, and may even result in weak game AI. Due to the complex nature of modern video games, the determination of a suitable evaluation function is often a difficult task. This paper discusses our work on fully automatically generating a good evaluation function for a real-time strategy (RTS) game.

The outline of the paper is as follows. In section 2, we discuss two approaches of creating adaptive game AI for RTS games. Section 3 describes how we collect domain knowledge of an RTS game in a data store. How we establish an evaluation function for RTS games automatically, based on the collected data, is discussed in section 4. In section 5, we test the performance of the generated evaluation function and provide the experimental results. We discuss the results in section 6, and in section 7 provide conclusions and describe future work.

2 Approaches

A first approach to adaptive game AI is by incrementally changing game AI to make it more effective. The speed of learning depends on the learning rate. If the learning rate is low, learning will take a long time. If it is high, results will be unreliable. Therefore, an incremental approach is not very suitable for rapidly adapting to observations in a complex video game environment.

A second approach is by allowing computer-controlled players to imitate human players. This approach can be particularly successful in games that have access to the Internet and that store and retrieve samples of gameplay experiences [5]. For this approach to be feasible, a central data store of gameplay samples must be created. Game AI can utilise this data store for two purposes: (1) to establish an evaluation function for games, and (2) to be used as a model by an adaptation mechanism. We follow the second approach in our research.

Our experimental focus is on RTS games, i.e., simulated war games. Here, a player needs to gather resources for the construction of units and buildings. The goal of the game is to defeat an enemy army in a real-time battle. We use RTS games for their highly challenging nature, which stems from three factors: (1) their high complexity, (2) the large amount of inherent uncertainty, and (3) the need for rapid decision making [2]. In the present research, we use the open-source RTS game Spring. A Spring game is won by the player who first destroys the opponent's 'Commander' unit.

3 Data Store of Gameplay Experiences

We define a gameplay experience as a set of observable features of the environment at a certain point in time. To create a data store of gameplay experiences for the Spring environment, we start by defining a basic set of features that will play an essential role in the game. For our first experiments, we decided to use the following five features.

1. Number of units observed (maximum 5000) of each type (over 200),
2. Number of enemy units within a 2000u radius of the Commander,
3. Number of enemy units within a 1000u radius of the Commander,
4. Number of enemy units within a 500u radius of the Commander.
5. Percentage of the environment visible to the friendly player.

Spring implements a so-called 'Line of Sight' visibility mechanism to each unit. This implies that game AI only has access to feature data of those parts of the environment that are visible to its own units (illustrated in Figure 1). When the game AI's information is restricted to what its own units can observe, we call this an *imperfect-information environment*. When we allow the

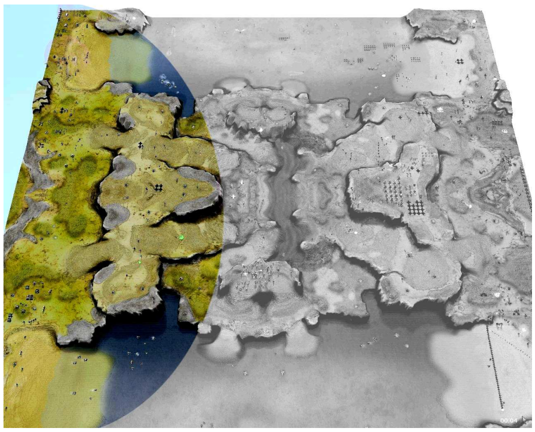


Fig. 1. Observation of information in a gameplay environment. Units controlled by the game AI are currently residing in the dark region. In an imperfect-information environment, only information in the dark region is available. In a perfect-information environment, information for both the dark and the light region is available.

game AI to access all information, regardless whether it is visible to its own units or not, we call this a *perfect-information environment*. We assume that the reliability of an evaluation function is highest when perfect information is used to generate it. For our experiments a data store was generated consisting of three different data sets: the first containing training data collected in a perfect-information environment, the second containing test data collected in a perfect-information environment, and the third containing test data collected in an imperfect-information environment.

4 Evaluation Function for Spring Games

This section discusses the automatic generation of an evaluation function for Spring. First, we discuss the design of an evaluation function based on weighted unit counts and tactical positions. Second, we discuss the application of temporal-difference (TD) learning [7] using a perfect-information data store to determine appropriate unit-type weights. Third, we discuss how the established evaluation function can be enhanced to enable it to deal with imperfect information.

4.1 Design of the Evaluation Function

The five defined features are selected as the basis of our evaluation function. Accordingly, the reference evaluation function for the game's status is denoted by

$$v = pv_1 + (1 - p)v_2 \qquad (1)$$

where $p \in [0...1]$ is a free parameter to determine the weight of each term of the evaluation function, the term v_1 represents the 'number of units observed of each type', and the term v_2 represents the 'number of enemy units within a certain radius of the Commander'. The term v_1 is denoted by,

$$v_1 = \sum_u w_u (c_{u_0} - c_{u_1}) \tag{2}$$

where w_u is the experimentally determined weight of the unit u, c_{u_0} is the number of units of type u that the game AI has, c_{u_1} is the number of units of type u that its opponent has. The term v_2 is denoted by:

$$v_2 = \sum_{r \in d} w_r (c_{r_1} - c_{r_0}) \tag{3}$$

where w_r is the experimentally determined weight of the radius r, c_{r_1} is the number of units the opponent has within a radius r of the game AI's Commander, c_{r_0} is the number of units the game AI has within a radius r of the opponent's Commander, and d is the set of experimentally determined radii $[500, 1000, 2000]$.

The evaluation function v can only produce an adequate result in a perfect-information environment, because in an imperfect-information environment c_{u_1}, c_{r_0} and c_{r_1} are unknown.

4.2 Learning Unit-type Weights

We used TD-learning to establish appropriate values w_u for all unit types u. TD-learning has been applied successfully to games, e.g., by Tesauro [8] for creating a strong AI for backgammon. Our application of TD-learning to generate an evaluation function for Spring is similar to its application by Beal and Smith [1] for determining piece values in chess.

Given a set W of weights $w_i, i \in \mathbb{N}, i \leq n$, and successive predictions P_t, the weights are updated as follows [1]:

$$\Delta W_t = \alpha (P_{t+1} - P_t) \sum_{k=1}^{t} \lambda^{t-k} \nabla_W P_k \tag{4}$$

where α is the learning rate and λ is the recency parameter controlling the weighting of predictions occurring k steps in the past. $\nabla_W P_k$ is the vector of partial derivatives of P_t with respect to W, also called the gradient of wP_k.

To apply TD-learning, a series of successive prediction probabilities (in this case: the probability of a player winning the game) must be available. The prediction probability of a game's status v_1, $P(v_1)$ is defined as

$$P(v_1) = \frac{1}{1 + e^{v_1}} \tag{5}$$

where v_1 is the evaluation function of the game's current status, denoted in Equation 2 (this entails that learning must be applied in a perfect-information environment). Figure 2 illustrates how a game state value v_1 of 0.942 is transformed into a prediction probability $P(v_1)$ of 0.72.

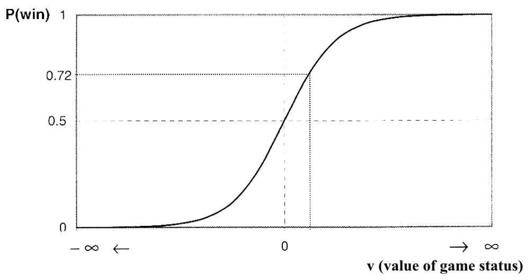


Fig. 2. Conversion from game status to prediction probability [1].

4.3 Dealing with Imperfect Information

In an imperfect-information environment, the evaluation value of the function denoted in Equation 1, which has been learned in a perfect-information environment, will typically be an overestimation. To deal with the imperfect information inherent in the Spring environment, we assume that it is possible to map reliably the imperfect feature-data to a prediction of the perfect feature-data. A straightforward enhancement of the function to this effect is to scale linearly the number of observed units to the non-observed region of the environment. Accordingly, the approximating evaluation function v' for an imperfect-information environment is as in Equation 1, with the term v'_1 denoted by

$$v'_1 = \sum_u w_u (c_{u_0} - \frac{o_{u_1}}{R}) \tag{6}$$

where w_u, c_{u_0} are as in Equation 2, o_{u_1} is the *observed* number of opponent's units of type u, $R \in [0, 1]$ is the fraction of the environment that is visible to the game AI. The term v'_2 is denoted by

$$v'_2 = \sum_{r \in d} w_r (\frac{o_{r_1}}{R_{r_1}} - \frac{o_{r_0}}{R_{r_0}}) \tag{7}$$

where w_r, r, d, and p are as in Equation 3, o_{r_1} is the *observed* number of units of the game AI within a radius r of the opponent's Commander, $R_{r_1} \in [0, 1]$ is the fraction of the environment that is visible to the opponent within the radius r, o_{r_0} is the *observed* number of units of the opponent within a radius r of the game AI's Commander, and $R_{r_0} \in [0, 1]$ is the fraction of the environment that is visible to the game AI within the radius r.

If enemy units are homogenously distributed over the environment, the approximating evaluation function applied to an imperfect-information environment will produce results close to those of the reference evaluation function in a perfect-information environment.

FRIENDLY TEAM	ENEMY TEAM	#GAMES IN TRAINING SET (Collected with perfect information)	#GAMES IN TEST SET (Collected with perfect information)	#GAMES IN TEST SET (Collected with imperfect information)
AAI	AAI (self-play)	500	200	200
AAI	TSI	100	200	200
AAI	CSAI	100	200	200
AAI	RAI	-	200	200

Table 1. The number of Spring games collected in the data store.

5 Experiments

This section discusses experiments that test our approach. We first describe the experimental setup and the performance evaluation, and then the experimental results.

5.1 Experimental Setup

To test our approach we start collecting feature data in the data store. For each player, feature data was gathered during gameplay. In our experiments we gathered data of games where two game AIs are posed against each other. Multiple Spring game AIs are available. We found one game AI which was open source, which we labelled 'AAI'. We enhanced this game AI with the ability to collect feature data in a data store, and the ability to disregard the line-of-sight visibility mechanism so that perfect information on the environment was available. As opposing game AIs, we used AAI itself, as well as three other AIs, namely 'TSI', 'CSAI', and 'RAI'. Table 1 lists the number of games from which we built the data store. The data collection process was as follows. During each game, feature data was collected every 127 game cycles, which corresponds to the update frequency of AAI. With 30 game cycles per second this resulted in feature data being collected every 4.233 seconds. The games were played on the map 'SmallDivide', which is a symmetrical map without water areas. All games were played under identical starting conditions.

We used a MatLab implementation of TD-learning to learn the unit-type weights w_u for the evaluation function of Equation 2. Unit-type weights were learned from feature data collected with perfect information from 700 games stored in the training set, namely all the games AAI played against itself (500), TSI (100), and CSAI (100). We did not include feature data collected in games where AAI was pitted against RAI, because we wanted to use RAI to test the generalisation ability of the learned evaluation function. The parameter α was set to 0.1, and the parameter λ was set to 0.95. Both parameter values were chosen in accordance with the research of Beal and Smith [1]. The unit-type weights were initialised to 1.0 before learning started.

Weights of the radii defined in the set d were chosen by the experimenter as 0.05, 0.20 and 0.75 for a radius of 2000, 1000 and 500, respectively. The parameter p was set to 1 (unit-based evaluation), 0 (evaluation of tactical positions) and 0.5 (a straightforward linear combination of the two terms).

5.2 Performance Evaluation

To evaluate the performance of the learned evaluation functions, we determined to what extent it is capable of predicting the actual outcome of a Spring game. For this purpose we defined the measure *absolute prediction* as the percentage of games of which the outcome is correctly predicted just before the end of the game. A high absolute prediction indicates that the evaluation function has the ability to evaluate correctly a game's status.

It is imperative that an evaluation function can evaluate a game's status correctly at the end of the game, and desirable throughout the play of a game. We defined the measure *weak relative prediction* as the game time at which the outcome of all tested games is predicted correctly at least 50% of the time. We defined the measure *normal relative prediction* and *strong relative prediction* as the game time at which the outcome of all tested games is predicted correctly at least 60% and 70% of the time, respectively. Since not all games last for an equal amount of time, we scaled the game time to 100% by averaging over the predictions made for each data point. A low relative prediction indicates that the evaluation function can correctly evaluate a game's status early in the game.

We determined the performance using two test sets. One test set contains feature data collected in a perfect-information environment, the other feature data collected in an imperfect-information environment. Feature data is collected of 800 games, where AAI is posed against itself (200), TSI (200), CSAI (200), and RAI (200).

We first tested the reference evaluation function in a perfect-information environment. Subsequently, we tested the approximating evaluation function in an imperfect-information environment.

5.3 Results of Learning Unit-type Weights

The Spring environment supports over 200 different unit types. During feature collection, we found that in the games played 89 different unit types were used. The TD-learning algorithm therefore learned weights for these 89 unit types. A summary of the results is listed in Table 2. Below, we give three observations on these results.

First, we observed that the highest weight has been assigned to the Advanced Metal Extractor. At first glance this seems surprising since it is not directly involved in combat situation. However, at the time the game AI destroys an Advanced Metal Extractor not only the opponent's ability to gather resources decreases, but it is also likely that the game AI has already penetrated its opponent's defences, since this unit typically is well protected and resides close to

UNIT-TYPE	WEIGHT
Advanced Metal Extractor *(Building)*	5.91
Thunder Bomber *(Aircraft)*	5.57
Metal Storage *(Building)*	4.28
Freedom Fighter *(Aircraft)*	4.23
Medium Assault Tank *(GroundUnit)*	4.18
...	...
Minelayer/Minesweeper with Anti-Mine Rocket *(GroundUnit)*	-1.10
Arm Advanced Solar Collector *(Building)*	-1.28
Light Amphibious Tank *(GroundUnit)*	-1.52
Energy Storage *(Building)*	-1.70
Defender Anti-air Tower *(Building)*	-2.82

Table 2. Learned unit-type weights (summary).

the Commander. This implies that destroying an Advanced Metal Extractor is a good indicator of success.

Second, we observed that some unit types obtained weights less than zero. This indicates that these unit types are of little use to the game AI and actually are a waste of resources. For instance, the Light Amphibious Tank is predictably useless, since our test-map contains no water.

Third, when looking into the weights of the unit types directly involved in combat, the Medium Assault Tank, Thunder Bomber and Freedom Fighter seem to be the most valuable.

5.4 Absolute-Prediction Performance Results

Using the learned unit-type weights, we determined the absolute-prediction performance of the evaluation functions. Table 3 lists the results for the trials where AAI was pitted against each of its four opponent AIs.

For the reference evaluation function in a perfect-information environment the average absolute-prediction performance is 76% for a unit-based evaluation ($p = 1$), 97% for an evaluation of tactical positions ($p = 0$), and 71% for a combined evaluation ($p = 0.5$). As the obtained absolute-prediction performances consistently predict more than 50% correctly, we may conclude that the reference evaluation function provides an effective basis for evaluating a game's status. Additionally, we observe that in two of the four trials the absolute-prediction performance of the combined evaluation is higher than that of the unit-based evaluation. However, on average the absolute-prediction performance of the combined evaluation is lower than that of the unit-based evaluation. These results imply that a combined evaluation of the defined terms has potential, yet it is currently in need of fine-tuning.

For the approximating evaluation function in an imperfect-information environment, the average absolute-prediction performance is 73% for a unit-based evaluation ($p = 1$), 92% for an evaluation of tactical positions ($p = 0$), and

AAI-AAI	ABS. PREDICTION PERFORMANCE	AAI-CSAI	ABS. PREDICTION PERFORMANCE
p=1	82% | 79%	p=1	85% | 87%
p=0	99% | 92%	p=0	98% | 95%
p=0.5	90% | 86%	p=0.5	86% | 87%
AAI-TSI		**AAI-RAI**	
p=1	68% | 57%	p=1	70% | 70%
p=0	96% | 92%	p=0	96% | 89%
p=0.5	56% | 50%	p=0.5	52% | 56%
AVERAGE			
p=1	76% | 73%		
p=0	97% | 92%		
p=0.5	71% | 70%		

Table 3. Absolute-prediction performance results. Each cell contains trial results for, in sequence, (1) the reference evaluation function applied in a perfect-information environment, and (2) the approximating evaluation function applied in an imperfect-information environment.

70% for a combined evaluation ($p = 0.5$). From these results, we may conclude that the approximating evaluation function in an imperfect-information environment successfully obtained an absolute-prediction performance comparable to the performance of the reference evaluation function in a perfect-information environment.

5.5 Relative-Prediction Performance Results

In Table 4 the relative-prediction performance is listed. We observe that for all values of p the weak relative-prediction performance is on average 54% in a perfect-information environment, and 51% in an imperfect-information environment. Results of the normal and strong relative-prediction performance are more divergent. The normal relative-prediction performances are on average 67% and 81% in a perfect-information and imperfect-information environment, respectively. The strong relative-prediction performances are on average 78% and 96% in a perfect-information and imperfect-information environment, respectively. This indicates that the reference evaluation function in a perfect-information environment manages to predict the outcome of a game considerably earlier than the approximating evaluation function in an imperfect-information environment.

As an exception to the before-mentioned indication, we observe that for $p = 1$ the obtained weak relative-prediction performance is significantly better in an imperfect-information environment than in a perfect-information environment. This might seem strange at first glance, but the explanation is rather straightforward: early in the game, when no units of the opponent have been observed yet, the approximating evaluation function will always predict a victory for the friendly team. Even though this prediction is meaningless (since it is generated without any information on the opponent), against a weaker game AI it is very likely to be correct, while against a stronger AI, it is likely to be false.

AAI-AAI	WEAK REL. PREDICITON PERFORMANCE	NORMAL REL. PREDICITON PERFORMANCE	STRONG REL. PREDICITON PERFORMANCE
p=1	28% | 1%	37% | 54%	47% | 99%
p=0	19% | 23%	59% | 65%	79% | 96%
p=0.5	32% | 50%	48% | 59%	64% | 73%
AAI-TSI			
p=1	85% | 1%	96% | 100%	100% | 100%
p=0	54% | 61%	73% | 91%	91% | 97%
p=0.5	97% | 100%	100% | 100%	100% | 100%
AAI-CSAI			
p=1	20% | 78%	26% | 83%	32% | 94%
p=0	92% | 92%	94% | 94%	95% | 97%
p=0.5	20% | 77%	26% | 83%	33% | 94%
AAI-RAI			
p=1	73% | 1%	89% | 57%	100% | 100%
p=0	27% | 33%	53% | 90%	95% | 96%
p=0.5	99% | 95%	100% | 100%	100% | 100%
AVERAGE			
p=1	52% | 20%	62% | 74%	70% | 98%
p=0	48% | 52%	70% | 85%	90% | 97%
p=0.5	62% | 81%	69% | 86%	74% | 92%

Table 4. Relative-prediction performance results. Each cell contains trial results for, in sequence, (1) the reference evaluation function applied in a perfect-information environment, and (2) the approximating evaluation function applied in an imperfect-information environment.

In a perfect-information environment, the strong relative-prediction performance is 70% on average for ($p = 0$), 90% on average for ($p = 1$), and 74% on average for ($p = 0.5$). We observe that only when a game is nearly finished, ($p = 1$) can accurately predict the game's outcome correctly. As ($p = 1$) obtained an *absolute*-prediction performance of 97%, this result implies that the term that evaluates tactical positions is particularly effective in the final phase of playing the game, and less effective in earlier phases.

Figure 3 displays the percentage of outcomes correctly predicted as a function over time. The figure compares the predictions of the reference evaluation function in a perfect-information environment, with the predictions of the approximating evaluation function in an imperfect-information environment, for $p = 0.5$.

6 Discussion

When evaluating exclusively by means of the feature 'number of units observed of each type' ($p = 1$), the reference evaluation function obtained an average

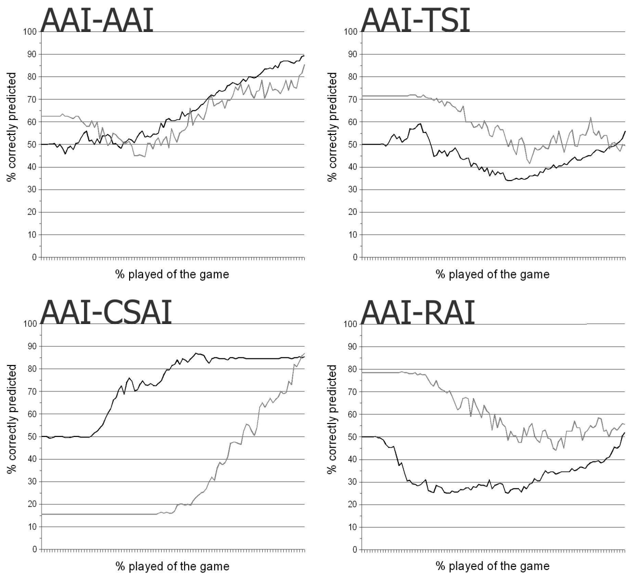


Fig. 3. Comparison of outcomes correctly predicted as a function over time. The black line represents the prediction performance of the reference evaluation function, the gray line that of the approximating evaluation function, for $p = 0.5$.

absolute-prediction performance of 76%. Thus, just before the game's end, it still wrongly predicts almost a quarter of the outcomes of the tested games. This is explained as follows. A SPRING game is not won by obtaining a territorial advantage, but by destroying the so-called Commander unit. Thus, even a player who has a material disadvantage may win if the enemy's Commander is taken out. Therefore, an evaluation function based purely on a comparison of material will not always be able to predict the outcome of a game correctly. Nevertheless, with a current average absolute-prediction performance of about 76%, and a strong relative-prediction performance of 70% on average, there is certainly room for improvement.

Evaluating tactical positions ($p = 0$) is likely to serve as such an improvement, as it obtained an average absolute-prediction performance of 97%. However, it also obtained a strong relative-prediction performance of 90% on average, which implies that it is not capable of correctly evaluating a game's status early in the game.

Straightforwardly combining of the two terms ($p = 0.5$) occasionally increased the absolute-prediction performance, but on average both the absolute- and relative-prediction performance deteriorated. We expect that including information on the phase of the game into the evaluation function will produce improved results. This enhancement will be the focus of future work.

7 Conclusions and Future Work

In this paper we discussed an approach to automatically generating an evaluation function for game AI in RTS games. From our experimental results, we may conclude that both the reference and approximating evaluation functions effectively predict the outcome of a Spring game expressed in terms of the absolute-prediction performance. The relative-prediction performance, which indicates how early in a game an outcome is predicted correctly, is lower (i.e., better) for the reference evaluation function than for the approximating evaluation function.

Two terms were defined to evaluate a game's status, namely a unit-based term and a term based on tactical positions. The term to evaluate tactical positions is capable of predicting the final outcome of the game almost perfectly. However, it only predicts accurately in the final phase of the game. The unit-based term, on the other hand, is only moderately accurate in predicting the final outcome of the game. However, it achieves a high prediction accuracy relatively early. This implies that the accuracy of outcome predictions is closely related to the phase of the game. Thus, to improve performance, the weights assigned to each term of the evaluation function should be made dependent on the phase of the game.

For future work, we will extend the evaluation functions with additional terms and will incorporate a mechanism to evaluate a game's status dependent on the phase of the game. Our findings will be incorporated in the design of an adaptation mechanism for RTS games.

References

1. Don F. Beal and Malcolm C. Smith. Learning piece values using temportal differences. *International Computer Chess Association (ICCA) Journal*, 20(3):147–151, 1997.
2. Michael Buro and Timothy M. Furtak. RTS games and real-time AI research. In *Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)*.
3. Pedro Demasi and Adriano J. de O. Cruz. Online coevolution for action games. *International Journal of Intelligent Games and Simulation*, 2(3):80–88, 2002.
4. Thore Graepel, Ralf Herbrich, and Julian Gold. Learning to fight. In Quasim Mehdi, Norman Gough, and David Al-Dabass, editors, *Proceedings of Computer Games: Artificial Intelligence, Design and Education (CGAIDE 2004)*.
5. Pieter H.M. Spronck. A model for reliable adaptive game intelligence. In David W. Aha, Hector Munoz-Avila, and Michael van Lent, editors, *Proceedings of the IJCAI-05 Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 95–100, 2005.
6. Pieter H.M. Spronck, Ida G. Sprinkhuizen-Kuyper, and Eric O. Postma. Online adaptation of game opponent AI with dynamic scripting. *International Journal of Intelligent Games and Simulation*, 3(1):45–53, 2004.
7. Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
8. Gerald Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.

Inducing and Measuring Emotion through a Multiplayer First-Person Shooter Computer Game

Paul P.A.B. Merkx[1], Khiet P. Truong[1] and Mark A. Neerincx[1,2]

[1] TNO Human Factors, P.O. Box 23, 3769ZG Soesterberg, The Netherlands
[2] Delft University of Technology, P.O. Box 5031, 2628CD Delft, The Netherlands
{paul.merkx,khiet.truong,mark.neerincx}@tno.nl

Abstract. To develop an annotated database of spontaneous, multi-modal, emotional expressions, recordings were made of facial and vocal expressions of emotions while participants were playing a multiplayer first-person shooter (fps) computer game. During a replay of the session, participants scored their own emotions by assigning values to them on an arousal and a valence scale, and by selecting emotion category labels (e.g. 'happiness' and 'frustration'). The fps-game proved to be a successful evoker of a diversity of emotions. The annotation results revealed interesting insights in current models of emotion. While the two-dimensional arousal-valence space is usually described as circular, we found a V-shape pattern of reported arousal and valence values instead. Furthermore, correlations were found between emotion characteristics and user-specific preferences about games. The recorded data will be used in future research concerning automatic and human recognition of emotions.

1 Introduction

Emotions and automatic emotion recognition increasingly play important roles in designing and developing computer games [16, 9]. Automatic emotion recognition is an emerging technology that can provide an enormous boost to the entertainment industry. Imagine an adaptive computer game that adjusts its difficulty level to the observed frustration level of the user [8]. Or imagine a speech-enabled computer game that responds to affective voice commands [13]. In order to develop affective videogames, the system first needs to be able to sense the player's emotional state in a least obtrusive manner. Physiological measures such as heart rate or skin response can be relatively good predictors of a human's emotional state but these are usually measured through special equipment that is attached to the body, which can be experienced as obtrusive by the player. Other channels through which emotions can be expressed are voice and facial expressions. Vocal and facial expressions of emotions can be relatively easily registered by a camera and a microphone which are usually perceived as less obtrusive. Hence, we focus on measurements of vocal and facial expressions in this study.

There is a vast amount of literature available on the automatic recognition of vocal and facial expressions of emotions (e.g., [23, 18]). However, it is difficult to compare these studies to each other and draw conclusions about the performances of the recognizers with respect to its applicability in a real-world situation. A reason for this, among others, is that there is no agreement on how to represent or describe emotions. Further, most of these studies are based on typical, full-blown emotions that were acted out by actors. We rarely encounter these typical extremities of emotions in real-life situations: in real-life, we tend to adhere to the unwritten social conversational rules and express more subtle emotions. One of the obstacles in emotion research is the lack of annotated (i.e., information about *what* type of emotion occured *when*) *natural* emotion data. For the development of automatic emotion recognition applications that employ machine learning techniques, a large amount of natural emotion data is needed to train and test the emotion models. Several eliciation methods have been employed in the past to evoke natural, spontaneous affective responses. For example, efforts have been made to elicit spontaneous emotions by showing movies or pictures (e.g., [15, 24]), by interacting with spoken-dialogue systems or virtual characters [1, 2, 6] and by playing games [14, 12, 22, 25]. Since most of the games are designed to evoke emotions and we think that games can trigger a relatively broad range of different types of emotions, we used a multiplayer first-person shooter computer game to elicit affective responses and to study what type of emotional experiences and expressions are associated with this type of computer game.

In this paper, we present our first results of this data collection effort and illustrate how computer games can be used to elicit affective responses. Vocal and facial expressions of the players were registered during gameplay and, subsequently, were evaluated on emotion by the players themselves. Two different annotation methods were used. In the first method, participants were asked to rate their own emotions on continuous scales of arousal (active vs. passive) and valence (positive vs. negative). In the second method, a categorical description of emotions was adopted and participants were asked to choose between a number of predefined emotion labels. They were also given the possibility to give their own emotion label if there was no appropriate label present.

This paper is organized as follows: Section 2 will present a short summary of the most important topics in current emotion research. Section 3 will describe how the experiment was accomplished, while Section 4 will outline its results. Finally, Section 5 will discuss how the results can be interpreted and used for future research.

2 Emotion Theory

There is an ongoing debate about the question how emotion can be defined and represented, especially concerning categorical and dimensional (or continuous) approaches to emotion. Research that supports the categorical approach to emotion usually asserts the existence of a number of basic emotions that are universally expressed and recognized (e.g., [17, 7]). The best known list of basic emotions is often termed the 'Big Six': happiness, sadness, fear, anger, surprise and disgust [5]. Corresponding with the categorical approach, emotions can be annotated by assigning category labels to them. The following twelve emotion labels were used in our experiment: happiness, surprise, anger, fear, disgust, excitement, amusement, relief, wonderment, frustration, boredom and malicious delight. This list of labels was based on basic emotions and typical game-related emotions (derived from [16]).

Research that supports the dimensional approach models emotion as a two- or three-dimensional space (e.g., [20, 4]). As an annotation method, participants can be asked to define a point in space that corresponds to a particular emotion. This point can be defined as the intersection of two or more values. Since consistently identifying a third dimension (e.g. tension, control or potency) proves to be difficult [21], two dimensions are usually used, which are labeled as arousal (ranging from 'calm' i.e., low arousal to 'excited' i.e., high arousal) and valence (the degree to which the pleasantness of an emotional experience is valued as negative or positive, thus ranging from highly negative to highly positive). An example of such a method is the Feeltrace annotation tool (see [4]). Although some agreement exists that reported values in arousal-valence space form a circular pattern, results in [3] describe annotation tasks where participants had to watch pictures or listen to sounds and where reported arousal-valence values formed a V-shaped pattern in the two-dimensional space.

In our experiment, the name of the arousal scale was changed to 'intensity'. This was done to make the task more accessible for naives, because using the old term might have yielded less consistent results due to the misunderstanding of the exact meaning of the term. In addition, there is no real appropriate Dutch translation of 'arousal' with respect to emotion annotation.

The next section describes how we applied the arousal-valence model and the category labels for the acquisition of an annotated emotion database.

3 Method

An experiment was performed in order to elicit emotional responses from participants who were invited to play a computer game and to carry out an experimental task.

3.1 Participants

Seventeen males and eleven females participated in the experiment with an average age of 22.1 years and a standard deviation of 2.8. They were recruited via e-mail under a selection criterium of age (between 18 and 30 years). People suffering from concentration problems were not allowed to participate in the experiment. In addition, participants were asked to bring a friend or relative in order to have a teammate who they were already acquainted with prior to the experiment. The reason for this is the fact that people report and show more arousal when playing with a friend compared to playing with a stranger [19], especially when friends are playing in the same room [11]. A compensation was paid to all participants, sometimes in combination with a bonus (which was rewarded to the players of the two teams with the highest scores in the game and to the two teams with the best collaborating teammates). The function of these bonusses was to keep the participants motivated and to encourage them to talk while playing since one of our goals was to elicit and record vocal expressions.

3.2 The Game

Participants played in two teams of two players. The game they played was the first-person shooter Unreal Tournament 2004 by Epic Games, which has the possibility for players to play the game with and against each other through a Local Area Network (LAN). Unreal Tournament 2004 offers diverse multiplayer game modes, from simple 'deathmatches' to rather complex mission-based modes. Since at least some participants were expected to be unfamiliar with the game, the game mode 'Capture the flag' was selected to be used in the experiment. This game mode has a low learning curve, but nevertheless invites strategy talk. A very small 3D world (called '1on1-Joust' in the game) was selected for the participants to play in, in order to evoke hectic situations in which (tactical) communication and frustrations would easily arise. For both teams the goal of the game was to capture the other teams flag as many times as possible while defending their own flag.

At any time in the game (for example, in less exciting periods in the game) the experimenter was able to generate unexpected in-game events to keep players interested and surprised. Some examples were the sudden appearence of monsters, the sudden ability to fly, an increasing or decreasing game speed and unexpected problems with the gameplay. These events were evoked at an average frequency of one event per minute using Unreal Tournament's build-in cheatcode list.

3.3 Apparatus

Four PC's (connected via a LAN) were used as game computers, while webcams and microphones were connected to four laptops, one for each participant. Video recordings were made using Logitech Quickcam Sphere webcams. In-game events were captured by creating a screenshot of the game every second. These screenshots were concatenated to one single avi file.

3.4 Procedure

In short, the time schedule for the experiment was as follows:

- General instruction (15 minutes)
- Training session: getting used to the game (10 minutes)
- Instruction about the computer task followed by a short break (20 minutes)

- Training session: getting used to the computer task (20 minutes)
- Session 1a: playing the game (20 minutes)
- First questionnaire followed by a break (25 minutes)
- Session 1b: computer task with a ten-minute break halfway. (50 minutes)
- Long break (40 minutes)
- Session 2a: playing the game (20 minutes)
- Second questionnaire followed by a break (25 minutes)
- Session 2b: computer task with a ten-minute break halfway. (50 minutes)

After reading a general instruction about the experiment, participants received a training session in which they could get used to the game rules and gameplay. During this session, which took ten minutes, the experimenter provided the players with some additional guidelines about the game, helping them to play and enjoy the game better. Beforehand, the experimenter reminded the players about the bonusses they could receive and stimulated them to discuss their strategies and to get a score as high as possible. Throughout the training session, recordings were made of players' faces and speech and of the in-game events.

Subsequently, participants received instructions about the experimental task they had to perform after each game (see Section 3.6). A thorough twenty-minute training session allowed them to ask questions about the task and to be corrected by the experimenter if necessary.

After they finished their training, this same procedure of playing and annotating was repeated two times while play time and consequently annotation time were twice as long. The schedule allowed for long breaks of twenty to thirty minutes between sessions and shorter breaks within sessions. Participants were also asked to fill in two short questionnaires between sessions (see Section 3.5).

3.5 Questionnaires

Participants performed a bipolar Big-Five personality test (see [10]) and a gaming-related questionnaire. The gaming-related questionnaire asked among others:

1. How much do you like to play games in general?
2. How much do you like to play first-person shooters in general?
3. Did you like to play the game in the experiment?
4. How much time do you play computer games in general?
5. How much time do you play computer games together with other people, for example using the internet?

3.6 The Annotation Task

The recorded material was presented to the participants only twenty minutes after playing, including continuous recordings of the in-game events. While watching and hearing their own facial and vocal expressions and the captured video

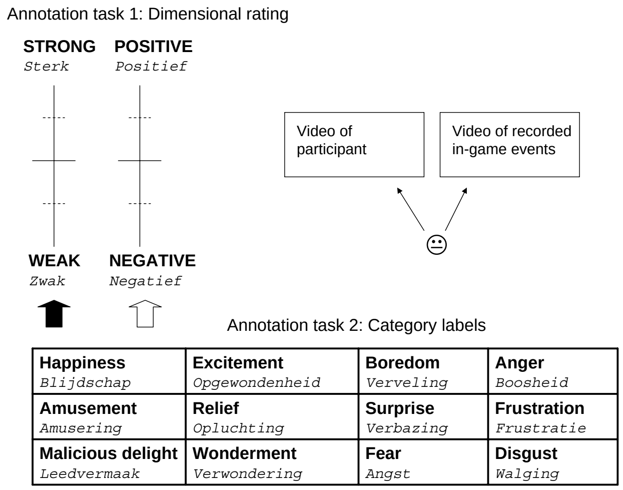


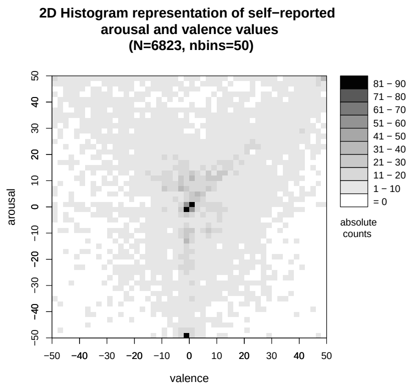
Fig. 1. The participant annotates his/her own emotions by watching his/her own video and the captured video stream from the in-game events. The participant performs two different annotation tasks. The Dutch translations that we used are in italics.

stream of the game, participants were asked to remember which emotions they experienced and how strong those emotions were at exactly that moment in the game. A two-part computer task was created for the annotation process. The first part asked participants about the intensity (ranging from weak to strong) and valence (reaching from negative to positive) of their emotions on two continuous scales, presented vertically and next to each other (as can be seen in Figure 1, together with Dutch translations, since the participants were Dutch). It is presumably easier for naive participants to perform the annotation on one scale at a time rather than evaluating both scales at the same time in a two-dimensional space, as is the case with the Feeltrace method [4]. Every ten seconds, an arrow indicated that a value on the intensity scale had to be selected, i.e. that a point on the first axis had to be clicked. Subsequently, a second arrow appeared, indicating that a value on the valence axis had to be clicked.

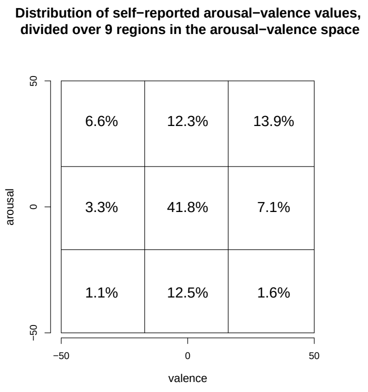
During the second part of the annotation task, participants watched and listened to the same material again and were asked how they would label their experienced emotions. This time, they were able to define the start and end points of the emotional moments themselves, activating or de-activating labels if a corresponding emotion occured or disappeared respectively. As previously mentioned, we selected twelve labels from which participants had to choose. Although the number of labels was kept as small as possible in order not to complicate the task for the participants, we expected that these labels would cover the whole range of emotions that could occur. In addition, participants had the possibility to verbally mention a term that they found more suitable for a certain emotion than the labels that the computer task contained. Figure 1 gives an overview of both the first part and the second part of the annotation task.

4 Results

Figure 2a gives a global overview of the reported values of all participants in the continuous annotation task. Darker areas contain more values than lighter areas. Figure 2b shows the percentage of reported values for nine subareas of the two-dimensional space. As can be derived from figures 2a and b, most values were selected from the center area of the two-dimensional space, while the upper-right corner with its high arousal and positive valence was another frequently visited region. In cases of low arousal, valence was nearly always analyzed as close to neutral, while high arousal was accompanied by more diverse valence values.



(a) 2D-histogram



(b) Density divided over 9 regions

Fig. 2. Graphs representing the density of reported arousal-valence values of all participants, plotted in an arousal-valence space.

We can observe in Figure 2 that the reported values follow a certain pattern, which can be described as a 'V-shape'. This is not entirely surprising since it is difficult to imagine an extremely negative or positive vocal or facial expression with weak intensity. Therefore, we calculated correlations between intensity and valence in order to see whether this observation can be supported. A significant quadratic relationship between arousal and valence was found with $r = 0.47$ (significant at $p < 0.01$). Furthermore, in the positive valence half of arousal-valence space, a linear relationship between arousal and valence with $r = 0.55$ ($p < 0.01$) was found. A corresponding negative correlation between arousal and valence was found in the negative valence half, with $r = -0.46$ ($p < 0.01$). These correlation figures suggest that higher intensity values are reported if valence is more extreme to either the positive or the negative side.

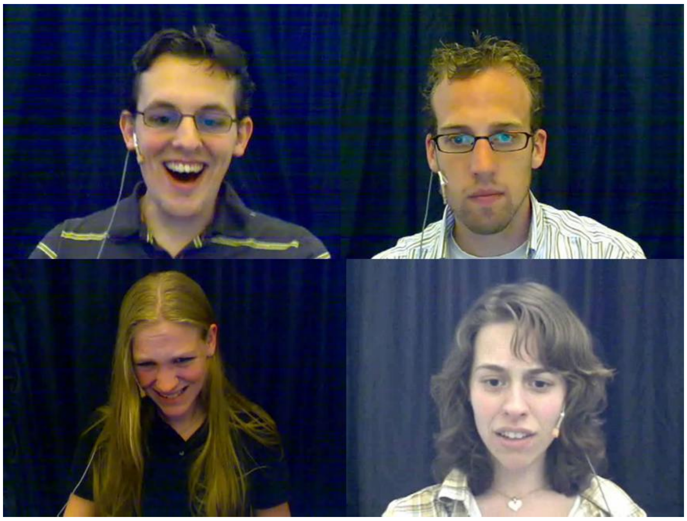
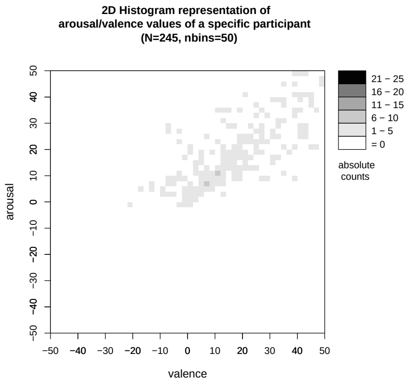
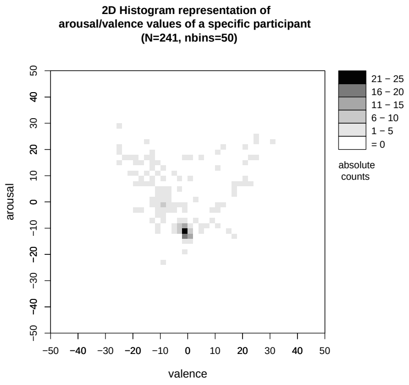


Fig. 3. Movie stills of registered emotions from four participants.



(a) an enthusiastic player



(b) a not so enthusiastic player

Fig. 4. 2D-histograms representing the density of reported arousal/valence values of two examples of different types of players, corresponding to the upper left and right players from Figure 3 respectively.

The diversity of emotions can also be attributed to the fact that different types of players participated in the experiment. Figure 3 shows some examples of movie stills of registered emotions from four participants. Figures 4a and b show plots of reported values in the two-dimensional space of an enthusiastic and an often frustrated not so enthusiastic player, corresponding to the upper left and right players from Figure 3 respectively (who also reported in the gaming-related questionnaire they liked and disliked the game respectively). The plots show again an overview of the reported values. It is clear that the reported values of these players still fall within the V-shape pattern, despite of individual differences between the players.

There seems to be a relation between reported valence values and the amount of everyday life multiplayer playing time (Pearson's $r = 0.49$; significant at $p < 0.05$) and on the question whether players like the first-person shooter genre in general (Spearman's $r = 0.50$; $p < 0.05$). The reported arousal values seem to be related with both the questions 'do you like to play games in general' (Spearman's $r = 0.50$; $p < 0.05$) and 'did you like to play the game in this experiment' (Spearman's $r = 0.45$; $p < 0.05$). In all of these cases, average arousal or valence values were higher if questions were answered more positively or if players played more often in everyday life.

5 Conclusions and Discussion

The most important aim of this study was to collect annotated natural emotion data through the use of a first-person shooter computer game and two different types of annotation methods. The first-person shooter proved to be a good elicitation tool for emotion. The game evoked a lot of different and intense emotions (ranging from negative to positive and from low to high arousal), probably caused by its intense atmosphere and the fact that the game was played by a group of four people who were all motivated to receive a reward. The game also evoked a lot of vocal expressions (in addition to facial expressions), probably caused by the fact that players were rewarded for good collaboration and the necessity of discussing strategies in order to perform better in the game. Making recordings proved to be relatively easy while participants were playing: they sat in front of their monitor while the webcam and the microphone registered all the expressions. In general, using a game as elicitation tool offers the opportunity to register relatively natural emotions in a somewhat laboratory environment: by playing computer games, participants become more immersed in a virtual world and pay less attention to their environment.

A first look at the annotation results offers some interesting insights considering the arousal-valence space. Our acquired arousal-valence values show that, while reported values in arousal-valence space are usually described as forming a circular pattern (e.g. [20, 4]), participants clearly followed a V-shape pattern while they were annotating their emotions. The V-shape pattern implies that participants interpreted a very negative or very positive emotion automatically as being accompanied by a high level of arousal and a more neutral emotion as having a lower arousal level. Note that the pattern of the reported arousal and valence values can be dependent on the type of game. However, our finding is in line with [3], which was interestingly the result of a very different emotion elicitation experiment. We can imagine that a very positive or very negative emotion is rather difficult to associate with low arousal. Future research might therefore focus on the validity of a circular or rectangular arousal-valence space. It might be interesting to investigate whether different emotion evokers yield different patterns of values in arousal-valence space. Further data analysis will reveal where naive participants think the emotion category labels from the second annotation task should be placed in the arousal-valence space.

Self-reported valence values were increasingly higher for players who spend more time on playing games with other humans in their spare time. This fact is a nice addition to what was concluded in [19], namely the fact that self-reported valence is more positive when people play against another human compared with playing against the computer. Hence, it appears that 'The people factor' is an important evoker of positive emotions in games, as was already mentioned in [16]. Furthermore, self-reported valence was also more positive if players liked the first-person shooter genre better. Arousal values were increasingly higher if players liked playing games in general or playing the game during the experiment. So it seems that in general, players experience more positive and/or more aroused emotions if they like a game or the multiplayer aspect of a game better.

In a follow-up study, participants will evaluate and annotate a selection of the recorded material in different modalities (visual and auditive). These annotations will be compared with the annotations made by the players themselves, trying to answer questions like 'How good can humans recognize emotions in different modalities (visual and auditive), with or without contextual information (the captured video stream of the in-game events) from which the emotions emerged?' and 'Are arousal and valence levels equally difficult to recognize in different modalities or do differences exist between them?'. Furthermore, the recorded data will be used for multimodal automatic emotion recognition research. Subsequently, we can investigate how human emotion recognition compares to automatic emotion recognition.

6 Acknowledgements

This study is supported by the Dutch BSIK project MultimediaN.

References

1. Ang, J., Dhillon, R., Krupski, A., Shriberg, E., Stolcke, A.: Prosody-based automatic detection of annoyance and frustration in human-computer dialog. Proceedings ICSLP (2002), 2037–2040
2. Batliner, A., Fischer, K., Huber, R., Spilker, J., Noth, E.: How to find trouble in communication. Speech Communication **40** (2003), 117–143
3. Bradley, M.M., Lang, P.J.: Affective reactions to acoustic stimuli. Psychophysiology **37** 2000, 204–215

4. Cowie, R., Douglas-Cowie, E., Savvidou, S., McMahon, E., Sawey, M., Schröder, M.: 'Feeltrace': An instrument for recording perceived emotion in real time. Proceedings of the ISCA Workshop on Speech and Emotion (2000), 19–24
5. Cowie, R., Cornelius, R.R.: Describing the emotional states that are expressed in speech. Speech Communication **40** 2003, 5–32
6. Cox, C.: Sensitive artificial listener induction techniques. Presented to HUMAINE Network of Excellence Summer School (2004), [Online Available: http://emotionresearch.net/ws/summerschool1/SALAS.ppt259]
7. Ekman, P. (1999).: Basic emotions. In Dalgleish, T., Power, M.: Handbook of cognition and emotion (1999), 45-60
8. Gilleade, K.M., Dix, A.: Using frustration in the design of adaptive videogames. Proceedings of ACE, Advances in Computer Entertainment Technology (2004) 228–232
9. Gilleade, K.M., Dix, A., Allanson, J.: Affective videogames and modes of affective gaming: Assist me, challenge me, Emote Me (ACE). Proceedings of DIGRA Conf. (2005)
10. Goldberg, L.R.: The development of markers for the Big-Five factor structure. Psychological assessment **4** 1992, 26–42
11. Halloran, J., Rogers, Y., Fitzpatrick, G.: From text to talk: Multiplayer games and voiceover IP. Proceedings of Level Up: First International Digital Games Research Conference (2003), 130–142
12. Johnstone, T.: Emotional speech elicited using computer games. Proceedings ICSLP (1996), 1985–1988
13. Jones, C.M., Sutherland, J.: Creating an emotionally reactive computer game responding to affective cues in speech. Proceedings of the 19th British HCI Group Annual Conference (2005)
14. Kim, J., Andre, E., Rehm, M., Vogt, T., Wagner, J.: Integrating information from speech and physiological signals to achieve emotional sensitivity. Proceedings Interspeech (2005), 809–812
15. Lang, P.J.: The emotion probe - studies of motivation and attention. American Psychologist **50** (1995), 371–385
16. Lazarro, N.: Why we play games: 4 keys to more emotion without story. Presented at the 2004 Game Developers Conference (2004)
17. Ortony, A., Turner, T.J.: Whats basic about basic emotions? Psychological Review **97**(3) (1990), 315–331
18. Pantic, M., Rothkrantz, L.J.M.: Automatic analysis of facial expressions: The state of the art. IEEE Transactions on Pattern Analysis and Machine Intelligence **22**(12) (2000), 1424–1445
19. Ravaja, N., Saari, T., Turpeinen, M., Laarni, J., Salminen, M., Kivikangas, M.: Spatial presence and emotions during video game playing: Does it matter with whom you play? Proceedings Presence (2005), 327–333
20. Russell, J.A.: A circumplex model of affect. Journal of personality and social psychology **39**(6) (1980), 1161–1178
21. Scherer, K.R.: What are emotions? And how can they be measured? Social Science Information **44**(4) 2005, 695–729
22. Yildirim, S., Lee, C.M., Lee, S.: Detecting politeness and frustration state of a child in a conversational computer game. Proceedings Interspeech (2005), 2209–2212
23. Ververidis, D., Kotropoulos, C.: Emotional speech recognition: Resources, features, and methods. Speech Communication **48**(9) (2006), 1162–1181

24. Wagner, J., Kim, J., Andre, E.: From physiological signals to emotions: implementing and comparing selected methods for feature extraction and classification. Proceedings IEEE ICME International Conference on Multimedia and Expo (2005), 940–943
25. Wang, N., Marsella, S.: Evg: an emotion evoking game. Proc. 6th International Conference on Intelligent Virtual Agents (2006)

Author Index