ELEKTROTEHNIŠKI VESTNIK 79(1-2): 35–40, 2012
ENGLISH EDITION

Detecting Fortresses in Chess

Matej Guid, Ivan Bratko

Artificial Intelligence Laboratory, Faculty of Computer and Information Science, University of Ljubljana, Tržaška c. 25, Ljubljana, Slovenia
Email: matej.guid@fri.uni-lj.si

Abstract. We introduce a computational method for semi-automatical detecting fortresses in the game of chess. It is based on computer heuristic search and can be easily used with any state-of-the-art chess program. We also demonstrate a method for avoiding fortresses and show how to find a break-through plan when one exists. Although the paper is not concerned with the question whether it is practical or not to implement the method *within* the state-of-the-art chess programs, the method can be useful, for example, in correspondence chess or in composing chess studies, where a human-computer interaction is of great importance, and the time available is significantly longer than in ordinary chess competitions.

Keywords: fortress, chess, computer chess, game playing, heuristic search

1 INTRODUCTION

In chess, *fortresses* are usually regarded as positions when one side has a material advantage, however, the defender's position is an impregnable fortress and the win cannot be achieved when both sides play optimally. The current state-of-the-art programs typically fail to recognise fortresses and seem to claim a winning advantage in such positions, although they are not able to achieve actually the win against adequate defence.

The position in Fig. 1 is taken from the book *Dvoretsky's Endgame Manual* [1]. The current state-of-the-art chess programs without an exception choose to take the black queen with the knight (1.Na4xb6), which leads to a big material advantage and to high evaluations that seemingly promise an easy win. However, it turns out that after 1...c7xb6 (black pawn takes the white knight) the backed-up evaluations, although staying high, cease to increase in further play. In fact, black position becomes an impregnable fortress and the win is no longer possible against an adequate defence.

Detecting fortresses is an unsolved task, at least in computer chess. It is possible (although not proven) that detection of fortresses in chess is nowadays possible by using Monte-Carlo Tree Search (MCTS) [2], [3], [4]. However, the state-of-the-art chess programs are based on heuristic search, and it seems rather impractical and even inefficient to supplement them with the MCTS algorithms for the purpose of detecting fortresses only. This is probably just another reason why currently the strongest chess programs are not able to detect fortresses such as the one shown in Fig. 1.

Received December 6, 2011
Accepted April 1, 2012

In this paper, we will introduce a novel method for detecting fortresses. It is based on computer heuristic search. We intend to demonstrate that due to lack of changes in backed-up heuristic evaluations between successive depths that are otherwise expected in won positions, fortresses can be detected effectively. We will also show how to avoid fortresses and possibly find a break-through plan when one exists.

2 DIRECTION-ORIENTED PLAY

The purpose of a heuristic evaluation function is to guide the game-tree search. Heuristic evaluation functions have to enable a program to find a *direction* of play towards a win, not only to maintain a won position. Backed-up heuristic values should in some way also reflect the progress towards the end of the game, and should therefore change as the search depth increases. Given a won position, if backed-up heuristic values remained the same with an increasing level of search, this would just ensure that the value "win" is maintained, without any guarantee of eventually winning, since the program would not be able to discriminate between a position with a slight advantage from the one that is clearly won.

This actually happens when one of the state-of-the-art chess programs is confronted with the simple task of winning the king and rook versus the lonely king endgame without the use of chess tablebases, and is limited with a sufficiently low search depth. It turns out that the program behaves in the following way: when using a search depth of 4 and higher, it assigns the same heuristic evaluations to all winning positions in this endgame (*i.e.*, the numerical value of 4.92),

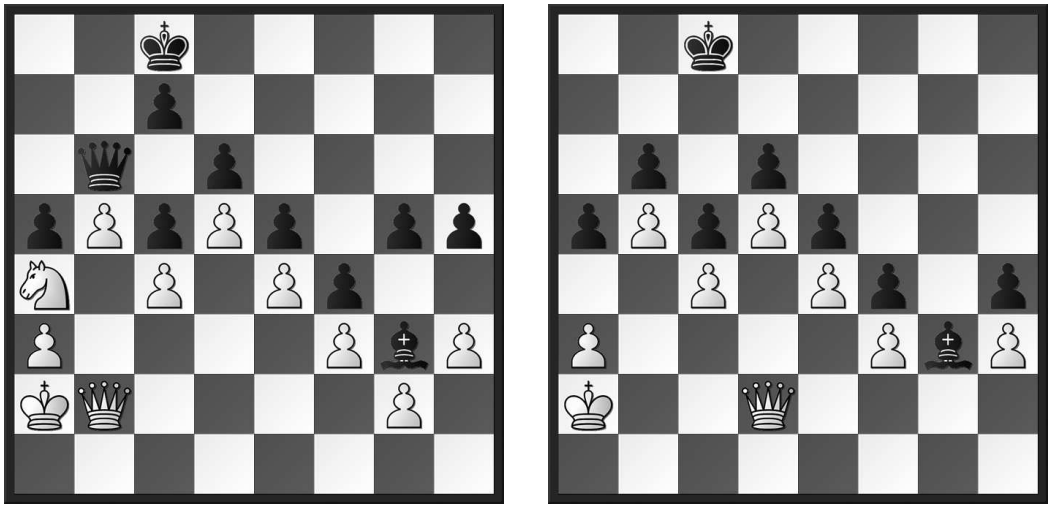

Figure 1. In the left side diagram, the white player is to move and has a winning positional advantage. State-of-the-art chess programs without any exception choose the move 1.Na4xb6 (white knight takes the black queen), which leads to a big material advantage. However, after 1...c7xb6 (black pawn takes the white knight) 2.h3-h4 (otherwise Black plays 2...h5-h4 with a draw) 2...g5xh4 3.Qb2-d2 h4-h3! 4.g2xh3 h5-h4 Black's position (see the diagram on the right side) becomes an impregnable fortress and the win is no longer possible against adequate defence. Nevertheless, as GM Dvoretsky indicates, white has a winning plan at disposal: Qb2-d2! followed by, Ka2-b3, Na4-b2, Kb3-a4, Nb2-d3-c1-b3. By executing this plan, White can gain the a5-pawn and win the game.

regardless of the search depth.* That is, although the program's evaluation function assesses positions in this endgame as won, it fails to distinguish between non-equally promising positions for achieving the final goal: delivering checkmate. This results in a rather ridiculous play by the winning side.

In 100 simulations mate-in-16 positions where the program played against the black player defending optimally (using tablebases), the program did not manage to deliver checkmate within prescribed 50 moves in several games, even at a 12-ply search. For example, at a 10-ply search the program did not win in 19 games, and the average length of the won games was 32 moves. Despite the fact that the program is aware of the 50-move rule†, it does not help it to always avoid the draw, when the depth of search is limited.

The same program, when using the shallow search of only 2 plies (*i.e.*, when the phenomenon does not occur), checkmates the opponent in 100% of the games played from randomly chosen positions, also finishing the task in considerably less moves on average. It is worth noting that the backed-up evaluations of the 2-ply search were on average increasing as the simulated games were proceeding, while at search depths where the phenomenon occurs they always stayed the same, unless the depth of search sufficed to find a principal variation that ended in checkmate (when the heuristic evaluation is no longer necessary).

3 How to detect fortresses?

Our proposed method for detecting fortresses is based on a very simple idea: if a position is a fortress, the side with a material advantage cannot demonstrate progress towards a win. Hence, backed-up heuristic evaluations obtained at various consecutive levels of search will *not* reflect the direction of the play towards a win, but will remain the same from a certain search depth on. Moreover, in such positions, several moves typically lead to the same directionless play and thus backed-up heuristic values of these moves should be the same or nearly the same from a certain depth of search on.

Figs. 2 and 4 illustrate the above point. The position in Fig. 2 is an elementary fortress [1]. White cannot overcome the barrier established on the squares between f8, f5, and h5, against an adequate defensive play by Black. Fig. 4 shows backed-up heuristic evaluations of the RYBKA chess program* at the levels of search in the range from 2 to 20 plies for the best five moves (or perhaps better: of the first five among several moves that lead to exactly the same backed-up heuristic evaluation value at a 20-ply search). Evaluations of none of the five moves reflect a progress towards a win. Moreover, the evaluations of all the five moves are practically the same

*The program where this phenomenon occurs is "RYBKA 2.1c 32-bit". In the year 2006, this was the highest rated chess program. The phenomenon no longer occurs in the later versions.

†The basic rules of the game according to FIDE say: "The game may be drawn if each player has made at least the last 50 consecutive moves without the movement of any pawn and without any capture."

*In the experiments, we used chess programs RYBKA and HOUDINI 1.5a. At the time of writing this paper, both programs are regarded as ones of the top chess programs.

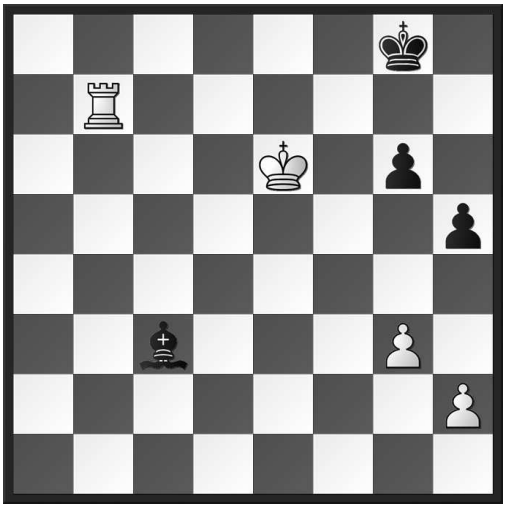

Figure 2. GM Dvoretsky: "This is an elementary fortress. White cannot overcome the barrier. If the black king returns to f4, Black takes g5 under control by means of Bf6. An advance of pawns brings no change."

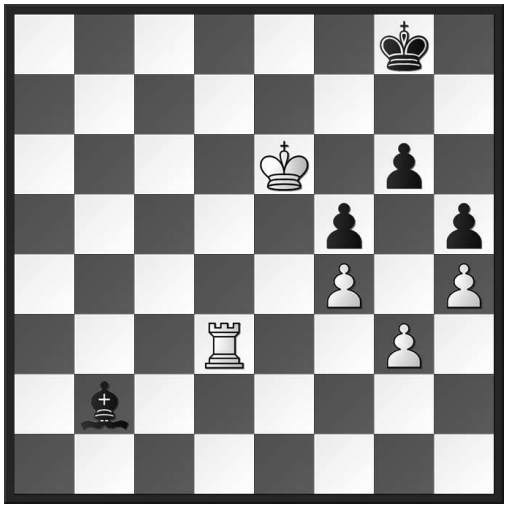

Figure 3. GM Dvoretsky: "It again seems as White cannot overcome the barrier, but in actuality, he can by means of a spectacular break-trough." White wins with 1.g3-g4! f5xg4 (1...h5xg4 2.h4-h5!) 2.f4-f5! Other plans are insufficient against correct defence.

from the search depth of 9 plies on. Such behaviour of a program indicates that the Black's position is an impregnable fortress.

4 How to avoid fortresses or find a break-through?

Let us return now to the position on the right side diagram in Fig. 1. White has a huge material advantage (the queen versus the bishop) and this material advantage is clearly reflected in backed-up evaluations of any heuristic-search-based chess program at any search depth. However, these evaluations remain practically the same at any search depth. With the Rybka chess program, for example, the backed-up evaluations remain within the interval [-6.80,-6.74] at all search depths in the range from 2 to 20. That is a clear indicator of an impregnable fortress. In order to avoid the fortress, the program should have foreseen this before grabbing the queen with the knight a few moves earlier (see the left side diagram in Fig. 1) and should have looked for alternative plans at that point.

Let us consider a different scenario. Suppose a heuristic-search-based chess program detects a potential fortress by using our proposed method. In the position in Fig. 3, the backed-up evaluations of the several highest ranked moves according to any state-of-the-art chess program remain practically the same at all levels of search up to 20 plies. How can a program find a possible break-through, providing that such a break-through exists? As GM Dvoretsky indicates, in the diagram of Fig. 3 White can win with a rather spectacular sacrifice of two of the three remaining white pawns [1]. Chess programs typically do not consider such a break-through as promising, since it involves losing material and consequently leads to inferior backed-up evaluations. Eventually, at sufficiently high search depths, the backed-up evaluations of the move that involves a sacrifice, also leading to successful break-through, would surpass the backed-up evaluations of the other moves. However, this usually occurs beyond search horizons of chess programs, and consequently the programs fail to find such a break-through.

In the diagrammed position in Fig. 3, the Rybka chess program does not recommend the break-through move 1.g3-g4 even up to the search depth of 30 plies. Only when set to return exact backed-up heuristic values of all possible moves in the diagrammed position, 1.g3-g4 becomes the program's first choice after reaching the search depth of 17 plies (see Fig. 5). Obtaining backed-up evaluations of all moves is much more time consuming compared to the normal behaviour of the program, and it is often not feasible under ordinary tournament conditions. Therefore, how can a break-through such as the one presented in this example be detected using a practical chess program?

We recommend the following procedure. When it becomes clear that the principal variation does not demonstrate a progress towards a win (*i.e.*, when the backed-up evaluations remain practically the same at all levels of search from a certain search depth on), it may be beneficial to analyse all possible moves up to some feasible search depth. If some move demonstrates increasing backed-up evaluations with an increasing level of search, such as the move 1.g3-g4 at the search

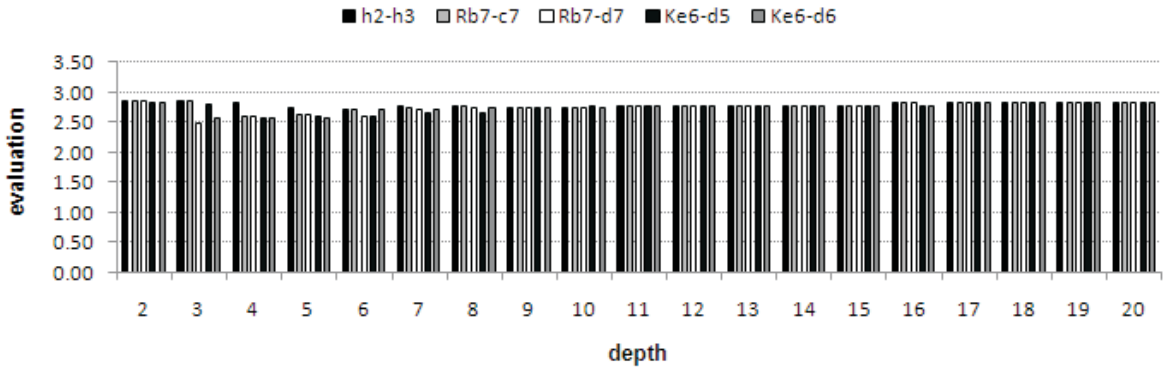

Figure 4. Backed-up heuristic evaluations of the RYBKA chess program at different levels of search in the range from 2 to 20 plies for the best five moves in Fig. 2 according to the program. The evaluations of other programs are qualitatively very similar.

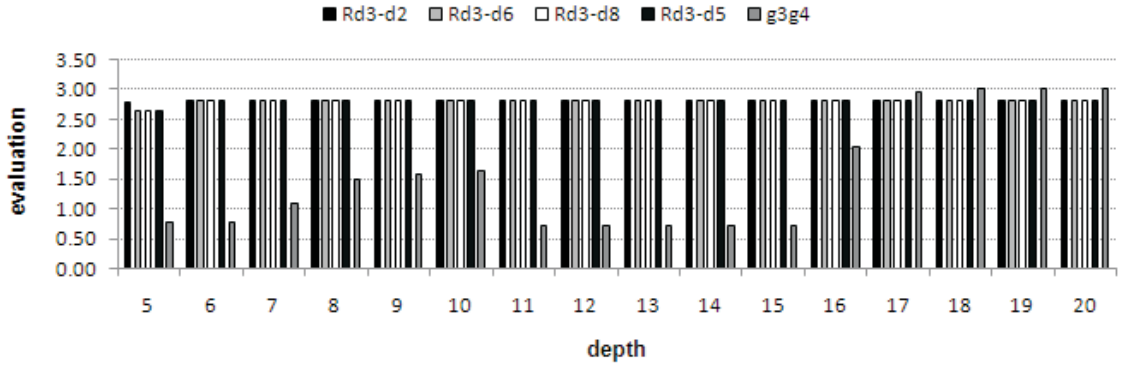

Figure 5. Backed-up heuristic evaluation of the RYBKA chess program for the position in Fig. 3 at different levels of search in the range from 2 to 20 plies, when the program is set to return exact backed-up heuristic values of all possible moves. The values for the first five program's choices at the highest depth of search are displayed. Under default settings, the program does not recommend 1.g3-g4 at any given level of search.

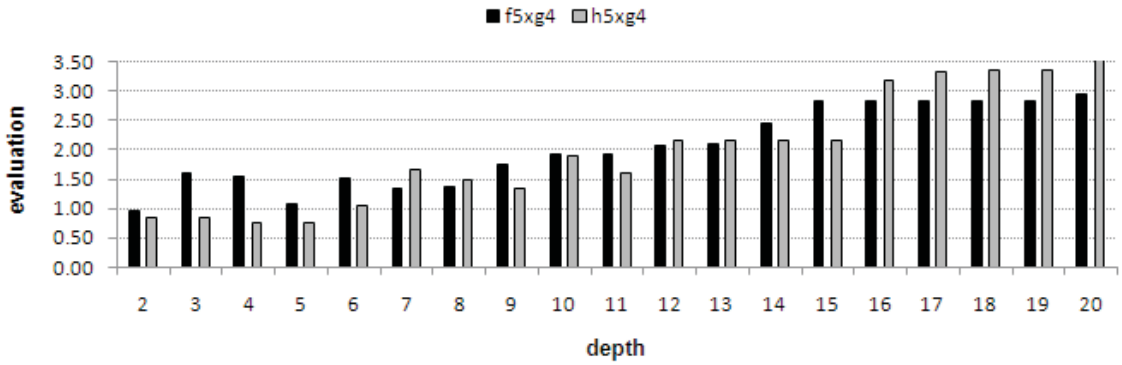

Figure 6. RYBKA's backed-up heuristic evaluations of two Black's best responses to the move 1.g3-g4 in Fig. 3 according to the program. They do not cease to increase with the search depth, indicating a successful break-through in what earlier seemed to be a fortress.

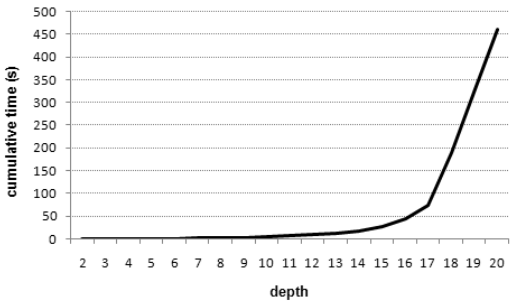

Figure 7. Cumulative time spent with an increasing search depth to obtain the backed-up heuristic evaluation of the RYBKA chess program for the position in Fig. 3 at different levels of search in the range from 2 to 20 plies, when the program was set to return exact backed-up heuristic values of all possible moves.

depths from 2 to 10 in the present example (see Fig. 5), it may be useful to devote more attention to such a move. Note that these shallow-depth evaluations may be significantly lower compared to those of the program's first choice.

Fig. 6 further illustrates this approach. We analysed two Black's responses to 1.g3-g4 that are the best responses according to the program, namely 1...f5xg4 and 1...h5xg4. Their backed-up evaluations do not cease to increase, and eventually become higher than the backed-up evaluations of the moves previously evaluated as the best.

Fig. 7 shows how the computer time increased with the depth when the program RYBKA was set to return exact backed-up heuristic values of all possible moves for the position in Fig. 3 (at 1.83 GHz and 2.0 GB RAM). This is of interest to see whether the proposed approach is feasible in practice. In the present case, the computer needed less than 10 seconds to analyse all the possible moves up to depth 12. Note that a 12-ply search may already suffice to detect a fortress (see Figs. 5 and 9). It is worth mentioning that fortresses typically occur with fewer pieces on board, and usually demand far less time for searching than regular middlegame positions.

5 AN EXPERIMENT WITH TWELVE FORTRESSES

We selected 12 positions from the aforementioned book that were recognised as fortresses by GM Dvoretsky [1]. The positions were analysed by the RYBKA and HOUDINI chess programs. The programs' backed-up evaluations of the search depths in the range from 2 up to 20 plies were obtained. Our claim was the following: backed-up evaluations in positions that could be regarded as fortresses will not behave as it is usual for winning positions, that is, they will not tend to increase with an increasing depth of search [5]. Four positions of the experimental data set are displayed in Fig. 8.

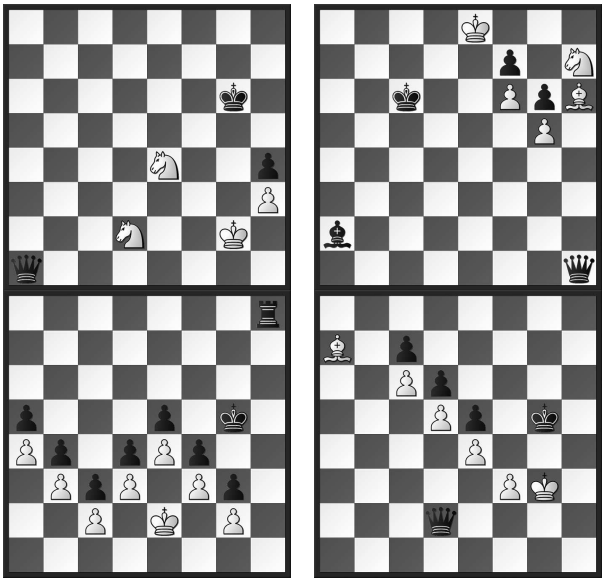

Figure 8. Positions of the white player are impregnable fortresses. Despite the huge material advantage of the black player in each of these positions, Black cannot win against an adequate defence by White.

The results of the experiment are demonstrated in Fig. 9, and confirm the above claim. For each of the twelve positions it holds that the backed-up evaluations remain practically the same from a certain search depth on, regardless of the program used. It is interesting to observe slight oscillations in the evaluations of HOUDINI. However, the demonstrated changes in the evaluations across several search depths are negligible compared to the expected changes in general of the backed-up heuristic values with the depth in won or lost positions (*e.g.*, compared to Fig. 6).

6 CONCLUSIONS

We introduce a novel idea for detecting fortresses in the game of chess. We demonstrate that a heuristic-search-based program is able to detect fortresses on the basis of backed-up values obtained at different levels of search. If a particular position is a fortress, the program is not able to show any progress towards a win and thus the backed-up values cease to change significantly from a certain search depth on. Moreover, it is rather typical of such positions that several alternative moves lead to the same (or rather similar) backed-up heuristic values at deeper levels of search.

We also demonstrate a possible way to avoid fortresses and how to find a break-through in a position that seems to be a fortress. Namely, when the backed-up evaluations remain practically the same at all levels of search from a certain search depth on, it may be

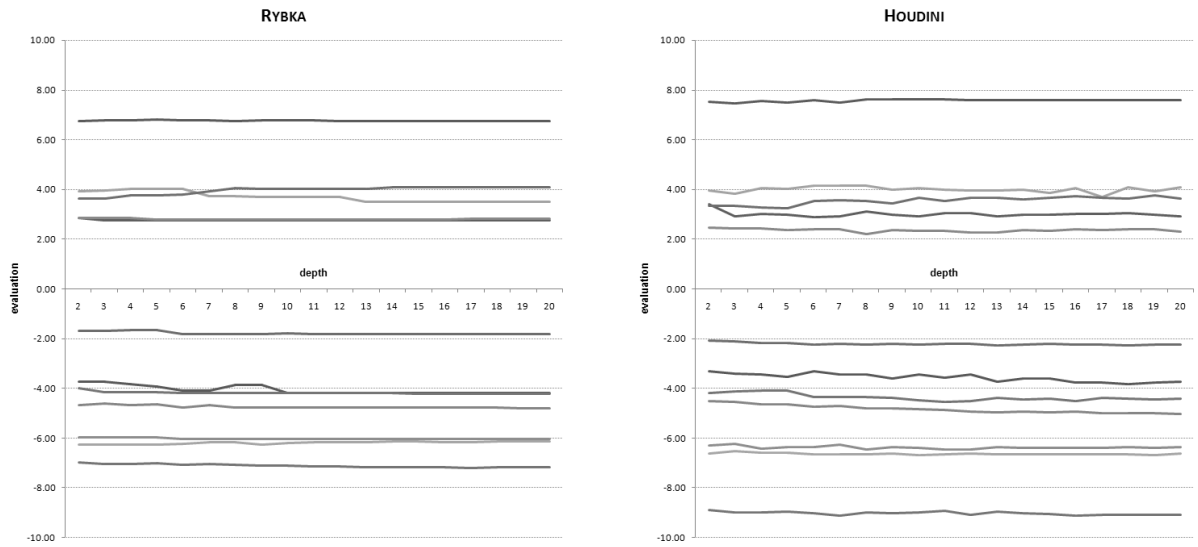

Figure 9. Backed-up heuristic evaluation of the RYBKA (left) and HOUDINI (right) chess programs for twelwe positions that were regarded as fortresses by GM Dvoretsky.

beneficial to analyse all possible moves up to some feasible search depth. If some move demonstrates positive changes in backed-up evaluations with an increasing level of search, it may be useful to devote more attention to such a move.

Future work may include a more detailed formulation of the algorithm for detecting fortresses, its implementation in a chess program, and its evaluation on a statistically significant set of test cases. Also, more empirical evidence would be helpful in order to determine whether the proposed method can be afforded by chess programs under various time constraints in a tournament play.

However, the method as presented in this paper can already be useful, for example, in correspondence chess or at composing chess studies, where a human-computer interaction is of great importance, and the time available is significantly larger than in ordinary chess competitions. In the era of strong chess engines, one of the important roles of a human in correspondence chess is to guide the engine(s) to the most promising continuations. One of the conclusions of this paper useful for competitors in correspondence chess is that a certain continuation may not be winning, when the backed-up evaluations, although staying high, cease to increase in further play.

The findings presented in this paper also represent a contribution to the understanding of the computer heuristic search in general. Namely, if backed-up heuristic evaluations of seemingly promising alternatives (*e.g.*, for solving a particular problem) cease to increase between successive depths of search, such alternatives may not be promising at all – even if the corresponding backed-up heuristic values obtained by heuristic search are extremely high.

REFERENCES

[1] M. Dvoretsky, *Dvoretsky's Endgame Manual, 2nd edition*. Russell Enterprises, Inc., 2008.

[2] L. Kocsis, C. Szepesvári, *Bandit based Monte-Carlo Planning*. *The European Conference on Machine Learning*, pp. 282–293, Springer, 2006.

[3] M.H. Winands, Y. Björnsson, J. Saito, *Monte-Carlo Tree Search Solver*. *Computers and Games, Lecture Notes in Computer Science*, vol. 5131, Springer, 2008.

[4] R. Coulom, *Efficient selectivity and backup operators in Monte-Carlo tree search*. *Computers and Games, Lecture Notes in Computer Science*, vol. 4630, Springer, 2007.

[5] M. Guid, *Search and Knowledge for Human and Machine Problem Solving*. Ph.D. Thesis, University of Ljubljana, 2010.

Matej Guid received his B.Sc. (2005) and Ph.D. (2010) degrees in computer science at the University of Ljubljana, Slovenia. He is a researcher at the Artificial Intelligence Laboratory, University of Ljubljana. His research interests include heuristic search, computer game-playing, automated explanation and tutoring systems, and argument-based machine learning. Chess has been one of his favorite hobbies since childhood. He was also a junior champion of Slovenia a couple of times, and holds the title of FIDE master.

Ivan Bratko received his B.Sc. (1970) and M.Sc. (1975) degrees in electrical engineering from the Faculty of Electrical Engineering, and the Ph.D. degree from the Faculty of Computer and Information Science (1978), all from the University of Ljubljana. He is professor at the Faculty of Computer and Information Science, University of Ljubljana. He is the head of Artificial Intelligence Laboratory since 1985; he also collaborates with Department of Intelligent Systems at the Jozef Stefan Institute. He is full member of the Slovenian Academy of Sciences and Arts (SAZU) since 2003, and member of *Academia Europaea* (since 2010). He received the ambassador in science award from Republic of Slovenia (1991), became a *Fellow of ECCAI* (*European Coordination Committee for Artificial Intelligence*) in 2001, and received the Zois award for outstanding scientific achievements from the state of Republic of Slovenia in 2007. His best known work is the book *Prolog Programming for Artificial Intelligence* (4th edition, Addison-Wesley 2011).