New Mathematics and Natural Computation

ENHANCED REALIZATION PROBABILITY SEARCH

MARK H.M. WINANDS
MICC-IKAT Games and AI Group, Faculty of Humanities and Sciences
Universiteit Maastricht, P.O. Box 616 6200MD Maastricht, The Netherlands
m.winands@micc.unimaas.nl

YNGVI BJÖRNSSON
Department of Computer Science, Reykjavík University
Ofanleiti 2 IS-103 Reykjavík, Iceland
yngvi@ru.is

In this paper we show that Realization Probability Search (RPS) significantly improves the playing strength of a world-class Lines-of-Action (LOA) computer program, even when used in combination with existing state-of-the-art $\alpha\beta$ search enhancements. In a 600-game match a RPS-based version of the program defeats the original one with a winning score of 62.5%. The main contribution of the paper, however, is the introduction of a much improved variant of RPS, called Enhanced Realization Probability Search (ERPS). The new algorithm addresses two weaknesses of RPS and overcomes them by using a better focussed re-searching scheme, resulting in both more robust tactical play and reduced search overhead. Our experiments in the domain of LOA show that ERPS offers just as significant improvement over regular RPS, as the latter improves upon regular search. More specifically, the ERPS-based variant scores 62.1% against the RPS variant, and an impressive 72.2% score against the original program. This represents an improvement of over 100 ELO points over the original state-of-the-art player.

Keywords: Search; heuristics; games.

1. Introduction

The alpha-beta ($\alpha\beta$) algorithm[14] is the standard search procedure for playing board games such as chess and checkers (and many others). The playing strength of programs employing the algorithm depends greatly on how deep they search critical lines of play. Therefore, over the years, many techniques for augmenting alpha-beta search with a more selective tree-expansion mechanism have been developed, so called *variable-depth search* techniques.[4] Promising lines of play are explored more deeply (search extensions), at the cost of others less interesting that are cut off prematurely (search reductions or forward pruning). Examples of widely used variable-depth search techniques are: *singular extensions*,[2] *null-move*,[9] *(multi-)probcut*,[8] *multi-cut*,[5] and *late-move reductions*.[20] Although all the aforementioned techniques aim at being domain independent, there are often specific search-space properties that make particular methods better suited than others for use in a given domain.

For example, the null-move technique has proven very successful in chess, whereas multi-probcut is the method of choice in Othello. Sometimes, new insights lead to ways for generalizing particular schemes such that they become applicable to a wider range of search domains.

One recent addition to the family of variable-depth search techniques is *Realization Probability Search (RPS)*, introduced by Tsuruoka *et al.*[23] in 2002. Using this technique his program, GEKISASHI, won the 2002 World Computer Shogi Championship, resulting in the algorithm gaining a wide acceptance in computer Shogi. Although researchers have experimented with the algorithm in other game domains with some success,[10] the question still remains how well the RPS enhancement works in other domains when used in a state-of-the-art game-playing program in combination with existing search enhancement schemes.

In this paper we investigate the use of RPS in the game of *Lines of Action (LOA)*. The contributions of this work are twofold. First, we demonstrate the usefulness of the RPS scheme in the domain of LOA by successfully incorporating it into a world-class program, effectively raising the level of state-of-the-art game-play in that domain. Secondly, we introduce two important enhancements to the RPS algorithm to overcome problems it has with endgame positions. This improvement, which we label *Enhanced Realization Probability Search (ERPS)*, greatly improves the effectiveness of RPS in our game domain. We conjecture that the enhancements may offer similar benefits in other game domains as well.

The paper is organized as follows. Section 2 briefly explains the RPS algorithm. In Section 3 we experiment with the RPS algorithm in the domain of LOA, and highlight its strengths and weaknesses. In Section 4 we introduce Enhanced Realization Probability Search, and we empirically evaluate it in Section 5. Finally, Section 6 gives conclusions and an outlook for future research.

2. Realization Probability Search

In this section we explain Realization Probability Search (RPS).[23] Move categories and fractional-ply extensions are explained in Subsection 2.1, probability-based fractional plies are explained in Subsection 2.2, and the RPS algorithm is outlined in Subsection 2.3.

2.1. *Fractional plies and move categories*

When making a move decision, game-playing programs typically search in an iterative fashion. From the current game position they start by searching one ply ahead, then two, and so on until the allotted time is up. The lookahead depth of each iteration, i.e., the numbers of plies, is called the *nominal* search depth. However, in each iteration most modern programs do not explore all lines of play to the exact nominal depth — some are explored more deeply whereas others are terminated prematurely.

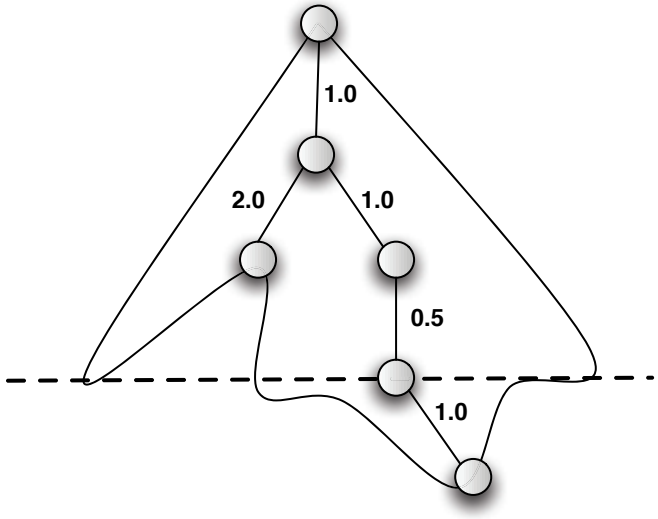

Fig. 1. Fractional-ply example.

A common framework for achieving such selectivity is the use of so-called *fractional-ply* extensions.[13,16] In this framework, instead of counting each move always as one ply towards the nominal search depth, moves are classified into different categories each with a different fractional ply-depth (or weight). This is illustrated in Figure 1. In the example there are three move categories, with weights 0.5, 1.0, and 2.0, respectively. The game tree is traversed in a left-to-right depth-first manner. Each line of play is expanded until the summed weights of the moves on the path from the root to the current node reaches (or exceeds) the nominal depth of 3.0. Because of the different weights, some lines of play are explored deeper than others. For example, in the figure the line of play to the left gets explored only 2-ply deep whereas the one to the right is explored 4-ply deep. This approach thus combines both search extensions (i.e., allows lines to be explored deeper than the nominal depth) and reductions (i.e., terminates lines before the nominal depth is reached) into a unified scheme. The name fractional-ply extensions, although an established term, is thus somewhat misleading. The reason for the name is that when the scheme was first introduced the fractional-plies where constrained to be less or equal to one, thus only resulting in extensions.

The moves are classified as belonging to different categories based on game-dependent features. For example, in chess, one category could be checking moves, another recaptures, etc. In other games different categories are used, for example, we describe the categories for LOA in Subsection 3.2. The categories are typically constructed manually. The same is true for the weights, although recently machine learning approaches have been used successfully to automatically tune them.[4,15]

2.2. *Realization probabilities*

The RPS algorithm is a new way of using fractional-ply extensions. The algorithm uses a probability-based approach to assign fractional-ply weights to move categories, and then uses re-searches to verify selected search results.

First, for each move category one must determine the probability that a move belonging to that category will be played. This probability is called the *transition probability*. This statistic is obtained from game records of matches played by expert players. The transition probability for a move category c is calculated as follows:

$$P_c = \frac{n_{played(c)}}{n_{available(c)}} \tag{2.1}$$

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

Originally, the *realization probability* of a node represented the probability that the moves leading to the node will be actually played. By definition, the realization probability of the root node is 1. The transition probabilities of moves were then used to compute the realization probability of a node in a recursive manner (by multiplying together the transition probabilities on the path leading to the node). If the realization probability would become smaller than a predefined threshold, the node would become a leaf. Since a probable move has a large transition probability while an improbable has a small probability, the search proceeds deeper along probable move sequences than improbable ones.

Instead of using the transition probabilities directly, we transform them into fractional plies as suggested by Tsuruoka *et al.*[23] The fractional ply FP of a move category is calculated by taking the logarithm of equation 2.1 in the following way:

$$FP = \frac{log(P_c)}{log(C)} \tag{2.2}$$

where C is a constant between 0 and 1. Experiments that we performed with our engine indicate that a value of 0.25 is a good setting for C in our game domain. Note that this setting is probably domain dependent, and a different value could be more appropriate in a different game or even game engine.

The fractional-ply values will be calculated off-line for all the different move categories, and used on-line by the search (as shown in Figure 1). We note that in the case where FP is larger than 1 it means the search is reduced while in the case FP is smaller than 1 the search is extended. By converting the transition probabilities to fractional plies, move weights now get added together instead of being multiplied. This has the advantage that we can use RPS alongside popular variable-search depth methods like null-move or multi-cut, which measure depth similarly.

2.3. *RPS algorithm*

In the previous subsection we saw that the *FP* values of the move categories are used to set the depth of the move to be explored. If one would naively implement this cut-off criterion simply based on the weights of the move categories one runs into difficulties because of the horizon effect. Move sequences with high *FP* values (i.e., low transition probability) get terminated quickly. Thus, if a player experiences a significant drop in its positional score as returned by the search, it is eager to play a possibly inferior move with a higher *FP* value, simply to push the inevitable score drop beyond its search horizon.

To avoid this problem, the algorithm is instructed to performs a *deeper* re-search for a move whose value is larger than the current best value (i.e., the α value). Instead of reducing the depth of the re-search by the fractional-ply value of the move (as is generally done), the search depth is decreased only by a small predefined *FP* value. In here we refer to it as *minFP*, and set it equal to the lowest move category value.

Apart from how the ply depth is determined, and the re-search, the algorithm is otherwise almost identical to NegaScout/PVS.[18,19] Figure 2 shows a C-like pseudo-code. Since the purpose of the preliminary search is only to check whether a move will improve upon the current best value, a null-window may be used.

3. RPS in LOA

In this section we investigate the application of RPS in LOA. First, we explain the game of LOA in Subsection 3.1; then, in Subsection 3.2, we shortly discuss the program MIA, and finally provide experimental evaluation of RPS in Subsection 3.3.

3.1. *LOA*

LOA (Lines of Action)[21] is a checkers-like game, but with a connection-based goal. The game is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The starting position is shown in Figure 3a. The players take turns moving a piece, starting with Black. A piece moves in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement (see Figure 3b). A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit (see Figure 3c). The connections within the unit may be either orthogonal or diagonal. In the case of simultaneous connection, the game is drawn. If a player cannot move, the player must pass. If a position with the same player to move occurs for the third time, the game is drawn.

Recently, LOA was used as a domain to test several new search and various other AI techniques.[3,12,25]

```
RPS(node, alpha, beta, depth){
    //Transposition table lookup, omitted
    ..................................
    if(depth == 0)
      return evaluate(pos);
    //Do not perform forward pruning in a potential principal variation
    if(node.node_type != PV_NODE){
      //Forward-pruning code, omitted
      ..................................
      if(forward_pruning condition holds) return beta;
    }
    next = firstSuccessor(node);
    while(next != null){
      alpha = max(alpha, best);
      decDepth = FP(next);
      //Preliminary Search Null-Window Search Part
      value = -RPS(next, -alpha-1, -alpha, depth-decDepth);
      //Re-search
      if(value > alpha)
        value = -RPS(next, -beta, -alpha, depth-minFP);

      if(value > best){
        best = value;
        if(best >= beta) goto Done;
      }
      next = nextSibling(next);
    }

    Done: //Store in Transposition table, omitted
    ..................................
}
```

Fig. 2. Pseudo code for Realization Probability Search.

3.2. *MIA*

MIA (Maastricht In Action) is a world-class LOA program, which won the LOA tournament at the eighth (2003), ninth (2004), and eleventh (2006) Computer Olympiad. It is considered the best LOA-playing entity in the world. All our experiments were performed in the latest version of the program, MIA 4.5.[a] The program is written in Java.

[a]The program and test sets can be found at: http://www.cs.unimaas.nl/m.winands/loa/.

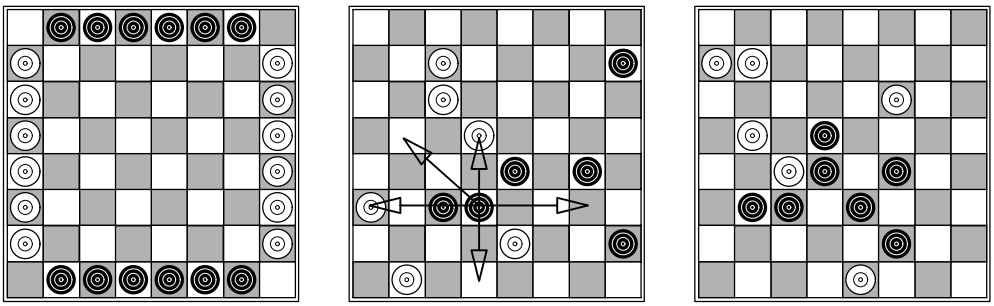

Fig. 3. (a) The initial position. (b) An example of legal moves. (c) Terminal position: Black wins.

MIA performs an $\alpha\beta$ depth-first iterative-deepening search in the PVS framework. A *two-deep* transposition table[7] is applied to prune a subtree or to narrow the $\alpha\beta$ window. At all interior nodes that are more than 2 ply away from the leaves, it generates all moves to perform Enhanced Transposition Cutoffs (ETC).[22] Next, a null-move[9] is performed adaptively.[11] Then, an enhanced multi-cut is performed.[5,27] For move ordering, the move stored in the transposition table (if applicable) is always tried first, followed by two killer moves.[1] These are the last two moves that were best, or at least caused a cutoff, at the given depth. Thereafter follow: (1) capture moves going to the inner area (the central 4×4 board) and (2) capture moves going to the middle area (the 6×6 rim). All the remaining moves are ordered decreasingly according to the relative history heuristic.[28] At the leaf nodes of the regular search, a quiescence search is performed to get more accurate evaluations. For additional details on the search engine and the evaluation function used in MIA, we refer to the Ph.D. thesis *Informed Search in Complex Games*.[25]

RPS is applied in MIA in the following way. First, we classify moves as captures or non-captures. Next, moves are further sub-classified based on the origin and destination of the move's from and to squares. The board is divided into five different regions: the corners, the 8×8 outer rim (except corners), the 6×6 inner rim, the 4×4 inner rim and the central 2×2 board. Finally, moves are further classified based on the number of squares traveled away from or towards the center-of-mass. In total 277 move categories can occur in the game according to this classification. The *FP* values of the move categories are between 0.5 and 4.0 (inclusive).

Table 1 gives the *FP* values of Black's moves in the position depicted in Figure 3b. On the one hand, a move that puts a piece further away from the center-of-mass (e.g., **d3-b5**) or puts a center piece in a corner (e.g., **e4-h1**) has a relatively high *FP* value. This indicates that the moves are improbable and therefore only a shallow search has to be performed. On the other hand, moves capturing pieces in the opponent's center-of-mass (e.g., **d3xd5** or **c3xc6**) have a low *FP* value. This indicates that the moves are promising and therefore a deeper search has to be performed.

Table 1. *FP* values of Black's moves in Figure 3b.

Move	FP value	Move	FP value	Move	FP value
h2-g1	1.90	c3xc6	0.95	d3-g3	2.33
h7-g8	3.10	c3-d2	1.57	d3xa3	4.00
g4xe2	1.22	c3-b4	3.14	g4-g3	1.80
g4-h3	3.39	d3-d1	3.73	g4-g5	1.80
g4-f5	1.97	d3xd5	0.78	h2-f2	1.70
h2-f4	1.12	e4xe2	1.50	c3-d4	1.79
e4-h1	4.00	e4-e6	3.30	c3-b2	3.14
h7-f7	1.70	e4-c4	4.00	h7-h5	1.67
d3-b5	4.00	c3-f3	1.43	h2-h4	1.90

3.3. *Experiments*

In the first experiment we tested how RPS affects the program's playing strength.[b] Two versions of MIA were matched against each other, one using RPS and the other not (called "Classic"). The programs are otherwise identical, and use all the enhancements described in Subsection 3.2. The programs played 600 games against each other, playing both colors equally. They always started from the same standardized set of 100 three-ply positions.[3] The thinking time was limited to 10 seconds per move, simulating tournament conditions. To prevent the programs from repeating games, a small random factor was included in their evaluation function.

Table 2. 600-game match results.

	Score	Winning %	Winning ratio
RPS vs. Classic	375-225	62.5%	1.67

The results are given in Table 2. The RPS version outplayed the original program with a winning score of 62.5% of the available points. The winning ratio is 1.67, meaning that it scored 67% more points than the opponent, showing that RPS improves the playing strength of MIA significantly. This experiment confirms the findings of Hashimoto *et al.*[10]

In the next experiment, Classic and RPS were tested on a set of 286 forced-win LOA positions. The maximum number of nodes evaluated is 50,000,000. The results are given in Table 3. The first column gives the names of the algorithms, the second the number of positions solved, and the third and fourth the number of nodes searched and the time it took, respectively. In the second column we see that 186 positions are solved by Classic, 130 positions by RPS. In the third and fourth

[b]The experiments were run on an AMD Opteron 2.6 GHz processor.

column the number of nodes and the time consumed are given for the subset of 120 positions, which both algorithms are able to solve. We see that not only does Classic solve more problems, but for those problems that both algorithms solve it explores three times smaller trees (in half the time). This shows that Classic, although performing worse in game play, is still the better tactical endgame solver.

Table 3. Comparing the search algorithms on 286 test positions with a limit of 50,000,000 nodes.

Algorithm	# of positions solved (out of 286)	120 positions	
		Total # of nodes	Total time (ms.)
Classic	186	557,663,533	1,245,883
RPS	130	1,400,473,146	2,400,466

Also of interest in Table 3 is that RPS is searching faster, that is, it explores 20-30 percent more nodes per second (nps) than its Classic counterpart. There are two reasons for this. First, RPS performs more re-searches than Classic. Re-visited nodes are often faster to generate and evaluate because of cached information. For example in MIA incremental move generation is implemented. The complete move list will be generated only after we have tried the transposition-table move and killer moves first. Moreover, in the evaluation function of MIA computations of certain features are cached, which can be used in other (deeper) positions.[26] Second, RPS searches deeper and narrower than Classic.[10,24] The deeper the search goes, the fewer the pieces are on the board, and the faster the nodes are evaluated in MIA.

We would like to remark that the potential difference in nps between RPS and Classic may depend on the game domain and program implementation.

4. Enhanced Realization Probability Search

In the previous section we saw that RPS outplays Classic in actual game-play, despite it solving fewer of the positions in our tactical test-suit. The added positional understanding gained with a deeper nominal search depth (because of the pruning) does apparently more than compensate for the tactics the program overlooks (because of the pruning). In this section we introduce Enhanced Realization Probability Search (ERPS), an improved variant that is tactically much safer than regular RPS, while still maintaining its positional advantages.

We identify two problems with RPS:

(1) A move causing a fail-low (i.e., the value returned from the search is less or equal to the lower bound of our search window) will always be ignored. A tactical move belonging to a move category with a high reduction factor may be wrongly pruned away because of a horizon effect. In this case there is a risk that the best move will not be played.

(2) A move causing a fail-high (i.e., the value returned from the search is greater or equal to the upper bound of our search window) will always be re-searched with minimal *FP*. A weak move may be unnecessarily extended because of a horizon effect. In this case valuable computing resources are wasted, resulting in a shallower nominal search depth.

Our enhancements aim at overcoming the aforementioned problems with RPS. The first one is based on an idea proposed by Björnsson *et al.*;[6] they suggest that pruning should take place only after the first m moves have been searched (i.e., looking at the rank of the move in the move ordering). This principle is applied in recent pruning techniques, such as LMR and RankCut.[17] Thus, to alleviate the first of the aforementioned problems of RPS, we cap the reduced depth of the first m moves to a maximum value t. This avoids aggressive reductions for early moves.

The second enhancement improves on how re-searches are done. Instead of performing a re-search right away to a full depth, we first interleave a shallower intermediate re-search (using a depth reduction that is the average of the original reduce depth and *minFP*). If that re-search also results in a fail-high, only then is the full-depth re-search performed. We restrict these intermediate re-searches to cases where there is a substantial difference between the original and full re-search depths (the averaged decrease depth is larger than a predefined threshold *delta*). This prevents insignificant explorations.

A pseudo code of ERPS is provided in Figure 4, providing the details of the implementation.

5. Experiments

In this section we empirically evaluate ERPS. The setup is similar to the earlier experiments: first the tactical strength is tested in Subsection 5.1, then the overall playing-strength in Subsection 5.2. In both cases we use parameter setting of (t=1, m=5, and *delta*=1.5), determined by trial and error.

5.1. *Tactical strength of ERPS*

In this experiment Classic and ERPS are tested on the same 286 forced-win LOA positions. The results are given in Table 4. In the first column the algorithms are mentioned. In the second column we see that 186 positions are solved by Classic, and 215 positions by ERPS. In the third and fourth column the number of nodes and the time consumed are given for the common subset of 174 positions that both algorithms solved. A look at the third column shows that ERPS search is a little bit more efficient than Classic in number of nodes explored. However, we see that ERPS is much faster than Classic in CPU time in the fourth column. For the same reasons as discussed in Subsection 3.3 ERPS generates faster nodes than Classic. Next, ERPS solves significantly more positions and is faster than Classic. As seen in Subsection 3.3 Classic was a superior solver compared to RPS. Thus, this suggests that ERPS search is a better endgame solver than RPS.

```
.....................................
    while(next != null){
      alpha = max(alpha, best);
      decDepth = FP(next);
      //Enhancement 1
      if(decDepth > t && moveCounter <= m)
            decDepth = t;
      //Preliminary Search Null-Window Search Part
      value = -RPS(next, -alpha-1, -alpha, depth-decDepth);
      //Re-search
      if(value > alpha){
       //Enhancement 2
        dec_depth = (minFP + dec_depth)/2;
        if(dec_depth>delta)
          value = -RPS(next, -alpha-1, -alpha, depth-decDepth);

        if(value > alpha)
          value = -RPS(next, -beta, -alpha, depth-minFP);
      }
      if(value > best){
        best = value;
        if(best >= beta) goto Done;
      }
      next = nextSibling(next);
    }
    .....................................
```

Fig. 4. ERPS: Two enhancements for Realization Probability Search.

Table 4. Comparing the search algorithms on 286 test positions with a limit of 50,000,000 nodes.

Algorithm	# of positions solved (out of 286)	174 positions	
		Total # of nodes	Total time (ms.)
Classic	186	1,224,042,920	2,848,431
ERPS	215	1,148,726,418	2,011,530

Since ERPS is a combination of two enhancements, we also tested the performance of each enhancement separately. The results are given in Table 5, where RPS-E1 indicates a RPS equipped with only the first enhancement and RPS-E2 a RPS equipped with only the second enhancement. Both enhanced variants of RPS are again compared to Classic. We see that RPS-E1 solved 175 positions and RPS-E2 183 positions, compared to 130 positions by RPS (as seen in Table 3). Thus,

both enhancements seem to be equally important. Moreover, the performance improvement of the enhancements seems to be additive because, as noted in Table 4, they solve 215 positions when used collectively.

Table 5. Comparing the search algorithms on 286 test positions with a limit of 50,000,000 nodes.

Algorithm	# of positions solved (out of 286)	152 positions	
		Total # of nodes	Total time (ms.)
Classic	186	911,024,108	2,097,086
RPS-E1	175	1,753,947,068	2,970,302
Algorithm	# of positions solved (out of 286)	156 positions	
		Total # of nodes	Total time (ms.)
Classic	186	963,640,832	2,181,385
RPS-E2	183	1,426,659,510	2,391,106

5.2. *Playing strength of ERPS*

In the last experiment we tested the playing strength of ERPS in actual game play. ERPS was matched against both the Classic version and the RPS version of MIA. All programs played under the same tournament conditions as used before. The results are given in Table 6. For a comparison the result of RPS against Classic is included as well.

The ERPS version outplays the original RPS version with a score of 62.1% (and a winning ratio of 1.63). This is similar level of improvement as RPS was over the Classic version. We also tested the ERPS version against the Classic version. ERPS, as expected, beats Classic with an even higher winning score (i.e., 72.2%). The results show that ERPS in a genuine improvement, significantly improving upon RPS.

Table 6. 600-game match results.

	Score	Winning %	Winning ratio
ERPS vs. RPS	372.5-227.5	62.1%	1.63
ERPS vs. Classic	433-167	72.2 %	2.59
RPS vs. Classic	375-225	62.5%	1.67

6. Conclusions and Future Research

In this paper we demonstrated the effectiveness of Realization Probability Search (RPS) in the game LOA, using a world-class LOA-playing program for our exper-

iments. We showed that RPS offers significant benefits, even when used in conjunction with other well-established state-of-the-art search enhancements. Furthermore, we introduced an improved variant of RPS: Enhanced Realization Probability Search (ERPS). The new algorithm is more robust tactically, resulting in much improved playing strength (in our test domain the improvement over regular RPS is of the same magnitude as the improvement RPS shows over regular search). A version of the LOA program enhanced with ERPS defeats the original LOA program by a winning score of 72.2%. This represents an improvement of over 100 ELO points, which is even more so impressive given that the original version was already playing at world-class level. The enhanced search variant may also offer benefits in other game-playing domains, as it does not rely on domain-specific knowledge.

As for future work, further refinement of the various parameters used in the ERPS algorithm may offer even further improvement, e.g., by allowing for a smoother transition when deciding to the decrease depth. The natural next step would be to test the ERPS algorithm in Shogi, a domain where RPS is already widely used. An even more aspiring goal would be to try it in chess. The search engines used in LOA programs are much similar to those used in a chess program, and share many of the same search enhancements. We have shown here that ERPS offers great benefits, even when used in conjunction with those other search enhancements. What makes this a somewhat more difficult task, is that additional work is required in defining a broad classification of possible move categories for chess.

Acknowledgments

This research was supported by grants from The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract number MIRG-CT-2005-017284. We would also like to thank Jahn-Takeshi Saito and Erik van der Werf for proof-reading the paper.

References

1. S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Annual Conference Proceedings*, pages 466–473. ACM Press, New York, NY, USA, 1977.
2. T.S. Anantharaman, M. Campbell, and F.-h. Hsu. Singular extensions: Adding selectivity to brute-force searching. *ICCA Journal*, 11(4):135–143, 1988. Also published (1990) in *Artificial Intelligence*, Vol. 43, No. 1, pp. 99–109.
3. D. Billings and Y. Björnsson. Search and knowledge in lines of action. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 231–248. Kluwer Academic Publishers, Boston, MA, USA, 2003.
4. Y. Björnsson. *Selective Depth-First Game-Tree Search.* PhD thesis, University of Alberta, Edmonton, Canada, 2002.

5. Y. Björnsson and T.A. Marsland. Multi-cut alpha-beta pruning. In H.J. van den Herik and H. Iida, editors, *Computers and Games*, volume 1558 of *Lecture Notes in Computing Science (LNCS)*, pages 15–24. Springer-Verlag, Berlin, Germany, 1999.
6. Y. Björnsson, T.A. Marsland, J. Schaeffer, and A. Junghans. Searching with uncertainty cut-offs. *ICCA Journal*, 20(1):29–37, 1997.
7. D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *ICCA Journal*, 19(3):175–180, 1996.
8. M. Buro. Experiments with multi-probcut and a new high-quality evaluation function for othello. In H.J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 77–96. Universiteit Maastricht, Maastricht, The Netherlands, 2000.
9. C. Donninger. Null move and deep search: Selective-search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143, 1993.
10. T. Hashimoto, J. Nagashima, M. Sakuta, J.W.H.M. Uiterwijk, and H. Iida. Automatic realization-probability search. Internal report, Dept. of Computer Science, University of Shizuoka, Hamamatsu, Japan, 2003.
11. E.A. Heinz. Adaptive null-move pruning. *ICCA Journal*, 22(3):123–132, 1999.
12. B. Helmstetter and T. Cazenave. Architecture d'un programme de lines of action. In T. Cazenave, editor, *Intelligence artificielle et jeux*, pages 117–126. Hermes Science, 2006. In French.
13. R.M. Hyatt. Crafty - chess program. 1996. ftp.cis.uab.edu/pub/hyatt.
14. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
15. L. Kocsis, Cs. Szepesvári, and M.H.M. Winands. Rspsa: Enhanced parameter optimisation in games. In H.J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H.H.M.L. Donkers, editors, *Advances in Computer Games Conference (ACG 2005)*, volume 4250 of *Lecture Notes in Computer Science (LNCS)*, pages 39–56. Springer-Verlag, Berlin, Germany, 2006.
16. D. Levy, D. Broughton, and M. Taylor. The sex algorithm in computer chess. 12(1):10–21, 1989.
17. Y.J. Lim and W.S. Lee. Rankcut - a domain independent forward pruning method for games. In *Proceedings of the AAAI 2006*, 2006.
18. T.A. Marsland. Relative efficiency of alpha-beta implementations. In *Proceedings of the 8^{th} International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 763–766. Karlsruhe, Germany, 1983.
19. A. Reinefeld. An improvement to the Scout search tree algorithm. *ICCA Journal*, 6(4):4–14, 1983.
20. T. Romstad. An introduction to late move reductions. 2006. www.glaurungchess.com /lmr.html.
21. S. Sackson. *A Gamut of Games*. Random House, New York, NY, USA, 1969.
22. J. Schaeffer and A. Plaat. New advances in alpha-beta searching. In *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, pages 124–130. ACM Press, New York, NY, USA, 1996.
23. Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):132–144, 2002.
24. Y. Tsuruoka, D. Yokoyama, T. Maruyama, and T. Chikayama. Game-tree search algorithm based on realization probability. In *Proceedings of Game Programming Workshop 2001*, pages 17–24, 2001.
25. M.H.M. Winands. *Informed Search in Complex Games*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2004.
26. M.H.M. Winands, H.J. van den Herik, and J.W.H.M. Uiterwijk. An evaluation func-

tion for lines of action. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *Advances in Computer Games 10: Many Games, Many Challenges*, pages 249–260. Kluwer Academic Publishers, Boston, MA, USA, 2003.

27. M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk, and E.C.D. van der Werf. Enhanced forward pruning. *Information Sciences*, 175(4):315–329, 2005.
28. M.H.M. Winands, E.C.D. van der Werf, H.J. van den Herik, and J.W.H.M. Uiterwijk. The relative history heuristic. In H.J. van den Herik, Y. Björnsson, and N.S. Netanyahu, editors, *Computers and Games*, volume 3846 of *Lecture Notes in Computer Science (LNCS)*, pages 262–272. Springer-Verlag, Berlin, Germany, 2006.