The Surakarta Bot Revealed

Mark H.M. Winands

Games and AI Group, Department of Data Science and Knowledge Engineering
Maastricht University, Maastricht, The Netherlands
m.winands@maastrichtuniversity.nl

Abstract. The board game Surakarta has been played at the ICGA Computer Olympiad since 2007. In this paper the ideas behind the agent SIA, which won the competition five times, are revealed. The paper describes its $\alpha\beta$-based variable-depth search mechanism. Search enhancements such as multi-cut forward pruning and Realization Probability Search are shown to improve the agent considerably. Additionally, features of the static evaluation function are presented. Experimental results indicate that features, which reward distribution of the pieces and penalize pieces that clutter together, give a genuine improvement in the playing strength.

1 Introduction

Since 2007 the board game Surakarta has been played six times at the ICGA Computer Olympiad, a multi-games event in which all of the participants are computer programs. The Surakarta agent SIA won the gold medal at the 12th, 13th, 15th, 17th, and 18th ICGA Computer Olympiad. It did not lose a single game in each tournament it participated.

In this paper the $\alpha\beta$-search based agent SIA is discussed in detail. It presents SIA's variable-depth search mechanism [9] that contains quiescence search [12], multi-cut forward pruning [2] and Realization Probability Search [13]. Also, the features of the static evaluation function are described and assessed.

The article is organized as follows. First, in Section 2 the game of Surakarta is briefly discussed. Next, SIA's $\alpha\beta$-search engine is introduced in Section 3. In Section 4 its variable-depth search mechanism is described. Subsequently, the evaluation function is proposed in Section 5. The experimental results are presented in Section 6. Finally, Section 7 gives conclusions and an outlook on future research.

2 Surakarta

Surakarta is a board game for two players (i.e., Black and White). It is played on a 6×6 board where eight loops extend out from it (see Fig. 1). The four small loops form together the inner circuit, whereas the four large loops form the outer circuit.

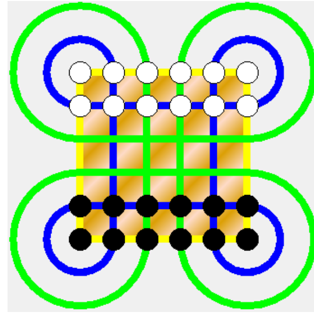

Fig. 1. Initial Surakarta position.

Players take turns moving one of their own pieces. In non-capturing moves, a piece travels – either orthogonally or diagonally – to a neighboring intersection. In a capturing move, a piece travels along a line, *traveling over at least one loop*, until it meets one of the opponent pieces. The captured piece is removed, and the capturing piece takes its place. The first player to capture all opponent's pieces wins. Draws can occur by repetition of moves or stalemate (cf. [6]). In this article, if a position with the same player to move occurs for the third time, the game is drawn. Additionally, if in the last fifty moves no capture was made, the game is scored as a draw as well.

Self-play experiments by SIA revealed that the game has an average branching factor of approximately 22 and an average game length of around 54 ply. The game-tree complexity is estimated to be about 10^{72}. Taking symmetry into account, its state-space complexity is 10^{15}.

3 SIA

SIA performs an $\alpha\beta$ depth-first iterative-deepening search in the PVS framework [10]. A *two-deep* transposition table [3] 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) [11]. 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 the capture moves. All the remaining moves are ordered decreasingly according to the relative history heuristic [16].

4 Variable-Depth Search

The $\alpha\beta$ algorithm [8] is still the standard search procedure for playing material-based board games such as chess and checkers. The playing strength of programs employing $\alpha\beta$ search 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 [9]. Promising lines of play are explored more deeply (search extensions), at the cost of other less interesting ones that are cut off prematurely (search reductions or forward pruning).

In the Surakarta engine SIA the following techniques are employed: *quiescence search* [7, 12], *multi-cut* [2], and *Realization Probability Search (RPS)* [13]. They are described in Subsections 4.1, 4.2, and 4.3, respectively.

4.1 Quiescence Search

When the $\alpha\beta$ search reaches the depth limit, a static evaluation function should be applied in the leaf node reached. This approach can have disastrous consequences because of the approximate nature of the evaluation function. Therefore a more sophisticated cut-off may be required. The evaluation function should only be applied to positions that are *quiescent*.

At the leaf nodes of the regular search, a quiescence search is performed to get more accurate evaluations. In SIA an extended version of quiescence search is implemented [12]. This type of a quiescence search limits the set of moves to be considered and uses the evaluations of interior nodes as lower / upper bounds of the resulting search value. As capture moves are responsible for swings in the evaluation function in Surakarta, only captures are considered for this part of the search.

4.2 Multi-Cut

Multi-cut pruning is a forward-pruning technique [2], which has been applied in chess and Lines of Action [15]. Before examining a node to full depth, the first M child nodes are searched to a depth reduced with a factor R. If at least C child nodes return a value larger than or equal to β, a cutoff occurs. However, if the pruning condition is not satisfied, the search continues as usual, re-exploring the node under consideration to a full depth d. In general the behavior of multi-cut is as follows. The higher M and R are and the lower C is, the higher the number of prunings is.

An enhanced version of multi-cut [15] is used in SIA. First, when at a reduced depth a winning value is found, the search is stopped and the winning value is returned. Second, if the multi-cut does not succeed in causing a cutoff, the moves causing a β-cutoff at the reduced depth are tried first in the normal search. Third, multi-cut is used in all nodes, except in the expected principal variation (so-called PV nodes). The idea is that it is too risky to prune forward there, because a possible mistake causes an immediate change of the principal variation. For all other nodes (so-called CUT and ALL nodes [9]), multi-cut is performed with the following parameter settings: $C = 3$ for a CUT node, $C = 2$ for an ALL node, $M = 10$ and $R = 2$ for both node types. The pseudo code in the PVS framework is given in Fig. 2.

```
.........................................................
//Forward-pruning code
 if(node.node_type != PV_NODE && depth > 2){
   next = firstSuccessor(node);
   c = 0, m = 0;
   while(next != null && m < M){
     value = -PVS(next, -beta, -alpha, depth-1-R);
     if(value >= beta){
       c++;
       //Keep track of the moves causing a cut-off at d-R
       storeCutOffNode(next);
       if(value >= WIN_SCORE)
         return value;
       else if(c >= C)
         return beta;
     }
     m++;
     next = nextSibling(next);
   }
   //Re-order moves
   putCutOffNodesInFront();
}
.........................................................
```

Fig. 2. Pseudo code for multi-cut

4.3 Realization Probability Search

One successful member of the family of variable-depth search techniques is *Realization Probability Search (RPS)*, introduced by Tsuruoka *et al.* [13] 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. It has been successfully applied in the Lines-of-Action engine MIA as well [14].

The RPS algorithm is an approach 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:

$$P_c \leftarrow \frac{n_{played(c)}}{n_{available(c)}} \tag{1}$$

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, they can be transformed into fractional plies [13]. The fractional ply *FP* of a move category is calculated by taking the logarithm of the transition probability in the following way:

$$FP \leftarrow log_K(P_c) \tag{2}$$

where K is a constant between 0 and 1. A value of 0.25 is a good setting for K in Surakarta. 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 are calculated off-line for all the different move categories, and used on-line by the search (as shown in Fig. 3 [14]). 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 RPS is used alongside multi-cut, which measures depth similarly.

However, setting the depth of the move based on its *FP* values 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, RPS is instructed to perform 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, called *minFP*. It is set 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 PVS [10]. Fig. 4 shows a C-like pseudo-code. Because 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.

RPS is applied in SIA in the following way. First, moves are classified as captures or non-captures. Next, moves are further subclassified based on the origin and destination of the move's from and to squares. The board is divided

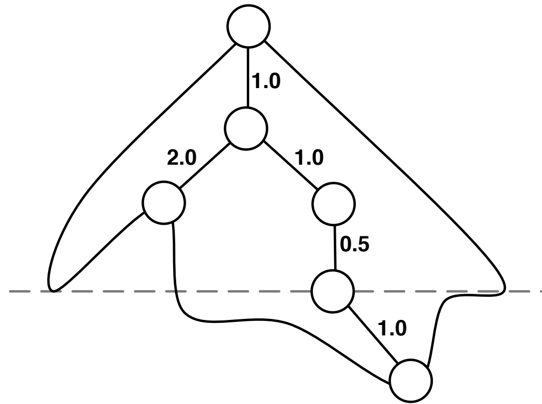

Fig. 3. Fractional-ply example for a nominal search depth of 3 [14].

into four different regions: the corners, the 6×6 outer rim (except corners), the 4×4 inner rim, and the central 2×2 board. In total 20 move categories can occur in the game according to this classification. The transition probabilities have been collected by letting SIA play 1000 games against itself. The final *FP* values of the move categories are capped between 0.5 and 4.0 (inclusive). They are shown in Table 1.

When looking at the transition probabilities, capture moves are in general preferred above non-capture moves. Although moving away from a corner is also strongly encouraged. Interestingly, when a move is a non-capture it is better to move towards the center. In case of a capture move, the opposite is true.

5 Evaluation Function

In this section the relevant features of the static evaluation function are enumerated and explained. The evaluator consists of the following five features: *material*, *mobility*, *player to move*, *quads*, and *distribution*. The choice of features that fully cover the description of a position is most relevant. It is better to have all features correct and all the initial weights wrong than to have the initial weights correct and miss one of the (important) features. The description of the features follows below; relevant examples and clarifications are given, adequate references to further details are supplied. It is followed by some information about the use of caching.

Material. Analogous to piece-square tables in chess, each piece obtains a value dependent on its board square in SIA. Especially, pieces at the corner are evaluated less. The relative values are given in the following matrix:

```
RPS(node, alpha, beta, depth){
    //Transposition table lookup, omitted
    ....................................
    if(depth <= 0)
      return quiescenceSearch(node, alpha, beta);
    //Do not perform forward pruning in a potential principal variation
    if(node.node_type != PV_NODE){
      //Multi-cut 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. 4. Pseudo code for Realization Probability Search.

$$\begin{bmatrix} 3 & 10 & 10 & 10 & 10 & 3 \\ 10 & 11 & 10 & 10 & 11 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 10 & 10 & 10 & 10 & 10 \\ 10 & 11 & 10 & 10 & 11 & 10 \\ 3 & 10 & 10 & 10 & 10 & 3 \end{bmatrix}$$

Mobility. Having more moves than the opponent may imply that you have more "freedom" that can be correlated with success. The computational requirements of the mobility feature are not high if only non-capture moves are considered. For each line configuration (represented as a bit vector) the mobility can be precomputed and stored in a table. During the search, the index scheme can be

Table 1. Move categories together with their transition probabilities and *FP* values.

Capture	Destination	Target	Transition Probability	FP value
No	Corner	Outer Rim	30.4%	0.85
No	Corner	Inner Rim	48.4%	0.52
No	Outer Rim	Corner	1.6%	2.97
No	Outer Rim	Outer Rim	12.9%	1.47
No	Outer Rim	Inner Rim	17.0%	1.27
No	Inner Rim	Corner	0.8%	3.45
No	Inner Rim	Outer Rim	6.7%	1.94
No	Inner Rim	Inner Rim	6.7%	1.95
No	Inner Rim	Center	11.5%	1.55
No	Center	Inner Rim	2.7%	2.60
No	Center	Center	7.4%	1.88
Yes	Outer Rim	Outer Rim	64.3%	0.50
Yes	Outer Rim	Inner Rim	59.0%	0.50
Yes	Outer Rim	Center	51.9%	0.50
Yes	Inner Rim	Outer Rim	63.4%	0.50
Yes	Inner Rim	Inner Rim	58.6%	0.50
Yes	Inner Rim	Center	49.4%	0.50
Yes	Center	Outer Rim	50.9%	0.50
Yes	Center	Inner Rim	47.2%	0.54
Yes	Center	Center	42.7%	0.61

updated incrementally and in the evaluation function only a few table lookups have to be done.

An advantage of this feature that it is fast to evaluate. A disadvantage of this implementation is that capture moves are not taken into account. This is *partially* mitigated by the quiescence search as only leaf nodes are evaluated that cannot start a capture sequence anymore. Still, it could be that the *non-moving* player has several possibilities to capture. Quiescence search is therefore not able to completely assess the capturing potential of one of the players.

Player to Move. The player-to-move feature is based on the basic principle of the initiative. It rewards the moving side. Having the initiative is mostly an advantage in Surakarta like in many other games.

Since SIA is using variable-depth search (because of quiescence search, the multi-cut, and RPS) not all leaf nodes are evaluated at the same depth. Therefore, leaf nodes in the search tree may have a different player to move, which is compensated in the evaluation function. This is done by giving a small bonus to the side to move.

Distribution. The distribution feature is based on the principle of spreading the pieces over the board to increase the potential to attack pieces of the opponent.

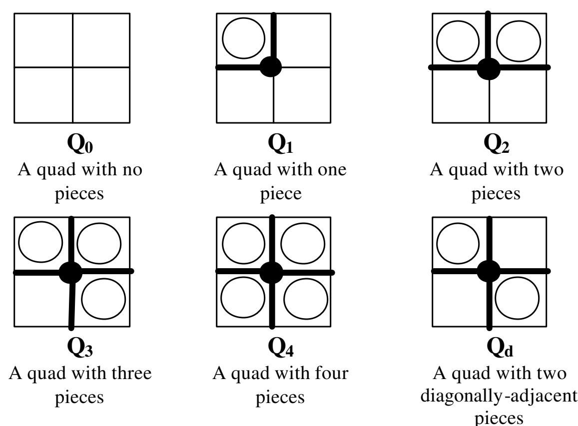

Fig. 5. Six different quad types.

In SIA this is done in a way which is primitive but effective. First the maximum number m of pieces of a player in a row or column is determined. The distribution is calculated as follows:

$$distribution = \frac{25 \times n}{max(2, m)} \tag{3}$$

where n is the number of pieces of a player. In such a way this feature prevents that there are too many pieces on one line. It is connected to the following feature, *quads*, that penalizes solid formations.

Quads. The quads feature prevents that pieces are cluttered together. The heuristic is based on the use of quads, an Optical Character Recognition method. A quad is defined as a 2×2 array of squares [5]. Taking into account rotational equivalence, there are six different quad types, depicted in Fig. 5. The values of each quad type is given in Table 2. Quads with 1 or 2 pieces receive a bonus, whereas quads with 4 pieces get a penalty.

Table 2. Quad values.

Quad types	Q_1	Q_2	Q_3	Q_4	Q_d
Values	5	5	0	-5	10

Caching Features. It is possible in SIA's evaluation function to cache computations of certain features, which can be used in other positions. The material, quads, and distribution features are independent of the position of the other side. They are stored in an evaluation cache table. In the current evaluation function this gives a speed-up of at least 30% in the number of nodes investigated per second.

6 Experiments

In this section the main components of SIA are tested. Different versions of SIA played at least 1000 games against each other, playing both colors equally. To prevent that games were repeated, a random factor was included in the evaluation function. Draws were considered half wins to each player to ensure the winning percentages sum to 100%. All experiments were performed on an Intel Xeon 5355 2.66 GHz computer. The engine has been implemented in Java. The remainder of this section is organized as follows. First, the variable-depth search techniques are tested in Subsection 6.1. Next, the features of the evaluation function are assessed in Subsection 6.2. Finally, SIA's performance on the ICGA Computer Olympiads is briefly discussed in Subsection 6.3.

6.1 Variable-Depth Search Experiments

In the first series of experiments SIA is instantiated using the various combinations of variable-depth search introduced in Section 4. A three-tuple $(RPS, Multi\text{-}Cut, QuiescenceSearch)$ is to represent the parameter setting used in each particular player instance. E.g., for the instantiation $\text{SIA}_{(off, multi, quiescence)}$, RPS is disabled, multi-cut and quiescence search are enabled.

For these experiments, the thinking time was limited to 5 seconds per move. The variable-depth search techniques were initially tested in an incremental way starting first with quiescence search, adding next multi-cut, and finally incorporating RPS. The first three rows of Table 3 show the results for them. It reveals that every search enhancement makes more or less the same contribution by increasing the winning percentage to approximately 70% for each addition. In the fourth row it was validated whether multi-cut does give an additional benefit to the RPS framework. By winning 63.5% of the games multi-cut is a genuine improvement. In the last row the results are given when SIA with all the enhancements played against the default fixed-depth version. All techniques combined lead to a 95% winning percentage. In the next experiment this combination is used.

6.2 Evaluation Function Results

In the last series of experiments four different evaluation functions competed with each other in a round-robin tournament. They are called MATERIAL, MOBILITY, DISTRIBUTION, and SIA. The MATERIAL evaluator consists out of the

Table 3. Winning percentage of testing various combinations of variable-depth search techniques. 95% confidence intervals are given.

		win %
$\text{SIA}_{(\mathit{off},\mathit{off},\mathit{quiescence})}$	$\text{SIA}_{(\mathit{off},\mathit{off},\mathit{off})}$	73.9 ± 1.5
$\text{SIA}_{(\mathit{off},\mathit{multi},\mathit{quiescence})}$	$\text{SIA}_{(\mathit{off},\mathit{off},\mathit{quiescence})}$	70.2 ± 1.4
$\text{SIA}_{(\mathit{RPS},\mathit{multi},\mathit{quiescence})}$	$\text{SIA}_{(\mathit{off},\mathit{multi},\mathit{quiescence})}$	75.3 ± 2.3
$\text{SIA}_{(\mathit{RPS},\mathit{multi},\mathit{quiescence})}$	$\text{SIA}_{(\mathit{RPS},\mathit{off},\mathit{quiescence})}$	63.5 ± 1.0
$\text{SIA}_{(\mathit{RPS},\mathit{multi},\mathit{quiescence})}$	$\text{SIA}_{(\mathit{off},\mathit{off},\mathit{off})}$	95.2 ± 0.8

piece-square table and a small random factor. The Mobility evaluator includes the former and incorporates the mobility and the player-to-move feature. Next, Distribution includes the distribution feature. Last, Sia adds the quads feature and represents the evaluation function discussed in Section 5. The weights of the features were partially tuned by TD-learning, partially manually. In these experiments, the thinking time was limited to 1 second per move.

The results of the round-robin tournament are given in Table 4. Each match data point represents the result of 1,000 games, with both colors played equally. The table shows that every added feature is a genuine improvement. Spreading the pieces over the board improves the performance of the play as the results of the Distribution and Sia evaluators indicate.

Table 4. Winning percentage of testing different evaluation functions. 95% confidence intervals are given. Each data point is based on a 1000-game match.

	Material	Mobility	Distribution	Sia
Material	-	42.9 ± 3.1	38.2 ± 3.0	32.2 ± 2.9
Mobility	57.1 ± 3.1	-	40.6 ± 3.0	35.8 ± 3.0
Distribution	61.8 ± 3.0	59.4 ± 3.0	-	46.7 ± 3.1
Sia	67.8 ± 2.9	64.2 ± 3.0	53.3 ± 3.1	-

6.3 Computer Olympiad Results

Since 2007 SIA has participated in the Surakarta tournaments at the 12th, 13th, 15th, 17th, and 18th ICGA Computer Olympiad. In the competition each agent receives 30 minutes of thinking time for the whole game, playing an equal number of games for each color. In these five tournaments SIA played a grand total of 32 games against 7 different opponents, winning all of them. This achievement is a validation of the approach to Surakarta proposed in this paper.

7 Conclusion and Future Research

This paper discussed the main components of the Surakarta agent SIA. Results showed that its variable-depth search mechanism improved the search considerably. Besides the classic quiescence search, multi-cut forward pruning and Realization Probability Search gave a boost in the game playing performance. Next, the evaluation function was described. Beside standard features such as material and mobility, features that helped to spread the pieces over the board gave a genuine increase in performance.

For future research adding a feature to determine who controls a circuit would lead potentially to an increase in playing performance. Next, endgame databases could help to improve the strength of the agent and ultimately help to solve the game. So far all endgame databases up to 8 pieces have been generated. Self-play results reveal that it takes on average 40 ply to reach them, which is too deep for a single search. If a 10-piece database or 12-piece database would be generated, it would take 34 or 30 ply, respectively. Larger databases would need several Terabytes of hard drive. An alternative is to use smaller databases and distribute the search over several cores as in done in Job-Level $\alpha\beta$ search [4].

Acknowledgments. Special thanks go to the anonymous referees whose comments helped to improve this paper.

References

1. Akl, S.G., Newborn, M.M.: The principal continuation and the killer heuristic. In: 1977 ACM Annual Conference Proceedings. pp. 466–473. ACM Press, New York, NY, USA (1977)
2. Björnsson, Y., Marsland, T.A.: Multi-cut alpha-beta pruning in game-tree search. Theoretical Computer Science 252(1-2), 177–196 (2001)
3. Breuker, D.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: Replacement schemes and two-level tables. ICCA Journal 19(3), 175–180 (1996)
4. Chen, J.C., Wu, I.C., Tseng, W.J., Lin, B.H., Chang, C.H.: Job-Level Alpha-Beta Search. IEEE Transactions on Computational Intelligence and AI in Games 7(1), 28–38 (2015)
5. Gray, S.B.: Local properties of binary images in two dimensions. IEEE Transactions on Computers 20(5), 551–561 (1971)
6. Handscomb, K.: Surakarta. Abstract Games 4(1), 8 (2003)
7. Kaindl, H., Horacek, H., Wagner, M.: Selective search versus brute force. ICCA Journal 9(3), 140–145 (1986)
8. Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)
9. Marsland, T.A., Björnsson, Y.: Variable-depth search. In: van den Herik, H.J., Monien, B. (eds.) Advances in Computer Games 9, pp. 9–24. Universiteit Maastricht, Maastricht, The Netherlands (2001)
10. Marsland, T.: A review of game-tree pruning. ICCA Journal 9(1), 3–19 (1986)
11. Schaeffer, J., Plaat, A.: New advances in alpha-beta searching. In: Proceedings of the 1996 ACM 24th Annual Conference on Computer Science, pp. 124–130. ACM Press, New York, NY, USA (1996)

12. Schrüfer, G.: A strategic quiescence search. ICCA Journal 12(1), 3–9 (1989)
13. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA Journal 25(3), 132–144 (2002)
14. Winands, M.H.M., Björnsson, Y.: Enhanced realization probability search. New Mathematics and Natural Computation 4(3), 329–342 (2008)
15. Winands, M.H.M., van den Herik, H.J., Uiterwijk, J.W.H.M., van der Werf, E.C.D.: Enhanced forward pruning. Information Sciences 175(4), 315–329 (2005)
16. Winands, M.H.M., van der Werf, E.C.D., van den Herik, H.J., Uiterwijk, J.W.H.M.: The relative history heuristic. In: van den Herik, H.J., Björnsson, Y., Netanyahu, N.S. (eds.) Computers and Games (CG 2004). Lecture Notes in Computer Science (LNCS), vol. 3846, pp. 262–272. Springer-Verlag, Berlin, Germany (2006)