Pruning playouts in Monte-Carlo Tree Search for the game of Havannah

Joris Duguépéroux, Ahmad Mazyad, Fabien Teytaud, Julien Dehos

Pruning playouts in Monte-Carlo Tree Search for the game of Havannah

Joris Duguépéroux, Ahmad Mazyad, Fabien Teytaud, and Julien Dehos

LISIC, ULCO, Université du Littoral Côte d'Opale

Abstract. Monte-Carlo Tree Search (MCTS) is a popular technique for playing multi-player games. In this paper, we propose a new method to bias the playout policy of MCTS. The idea is to prune the decisions which seem "bad" (according to the previous iterations of the algorithm) before computing each playout. Thus, the method evaluates the estimated "good" moves more precisely. We have tested our improvement for the game of Havannah and compared it to several classic improvements. Our method outperforms the classic version of MCTS (with the RAVE improvement) and the different playout policies of MCTS that we have experimented.

1 Introduction

Monte-Carlo Tree Search (MCTS) algorithms are recent algorithms for decision making problems [7, 6] . They are competitively used in discrete, observable and uncertain environments with a finite horizon and when the number of possible states is large. MCTS algorithms evaluate a state of the problem using a Monte-Carlo simulation (roughly, by performing numerous playouts starting from this state). Therefore, they require no evaluation function, which makes them quite generic and usable on a large number of applications. Many games are naturally suited for these algorithms so games are classically used for comparing such algorithms.

In this paper, we propose a method to improve the Monte-Carlo simulation (playouts) by pruning some of the possible moves. The idea is to ignore the decisions which seem "bad" when computing a playout, and thus to consider the "good" moves more precisely. We choose the moves to be pruned thanks to statistics established during previous playouts.

We experiment our improvement, called "Playout Pruning with Rave" (PPR) on the game of Havannah. Classic MCTS algorithms already provide good results with this game but our experiments show that PPR performs better. We also compare PPR to four well-known MCTS improvements (PoolRave, LGRF1, MAST and NAST2).

The remaining of this paper presents the game of Havannah in Section 2 and the Monte-Carlo Tree Search algorithms in Section 3. Our new improvement is described in Section 4. We present our results in Section 5. Finally, we conclude in Section 6.

2 Game of Havannah

The game of Havannah is a 2-player board game created by Christian Freeling in 1979 and updated in 1992 [26]. It belongs to the family of connection games with hexagonal cells. It is played on a hexagonal board, meaning 6 corners and 6 edges (corner stones do not belong to edges). At each turn a player has to play a stone in an empty cell. The goal is to realize one of these three shapes (i) a ring, which is a loop around one or more cells (empty or occupied by any stones) (ii) a bridge, which is a continuous string of stones connecting two corners (iii) a fork, which is a continuous string of stones connecting three edges. If there is no empty cell left and if no player wins then it is a draw (see Fig. 1). Previous studies related to the Monte-Carlo Tree Search algorithm applied to the game of Havannah can be found in [30, 20, 10].

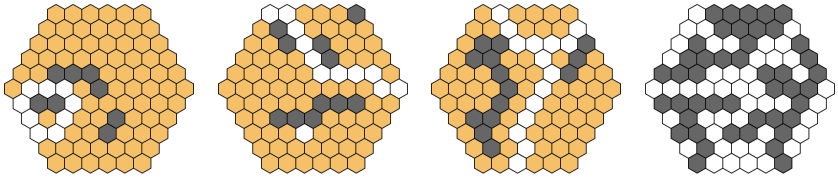

Fig. 1. The three winning shapes of Havannah (wins for the white player): a ring (left), a bridge (middle left) and a fork (middle right), and a draw (right).

3 Monte-Carlo Tree Search algorithms

The Monte-Carlo Tree Search (MCTS) algorithm is currently a state-of-the-art algorithm for many decision making problems [31, 3, 16, 9], and is particularly relevant in games [12, 21, 5, 1, 30, 22, 29, 19, 15, 14] . The general principle of MCTS is to iteratively build a tree and perform playouts to bias the decision making process toward the best decisions [18, 7, 6]. Starting with the current state s_0 of a problem, the MCTS algorithm incrementally builds a subtree of the future states. Here, the goal is to get an unbalanced subtree, where the branches with (estimated) good states are more developed. The subtree is built in four steps: *selection*, *expansion*, *simulation* and *backpropagation* (see Fig. 2).

The *selection* step is to choose an existing node among available nodes in the subtree. The most common implementation of MCTS is the Upper Confidence Tree (UCT) [18] which uses a bandit formula for choosing a node. A possible bandit formula is defined as follows:

$$s_1 \leftarrow \underset{j \in \mathcal{C}_{s_1}}{\arg\max} \left[\frac{w_j}{n_j} + K \sqrt{\frac{\ln(n_{s_1})}{n_j}} \right],$$

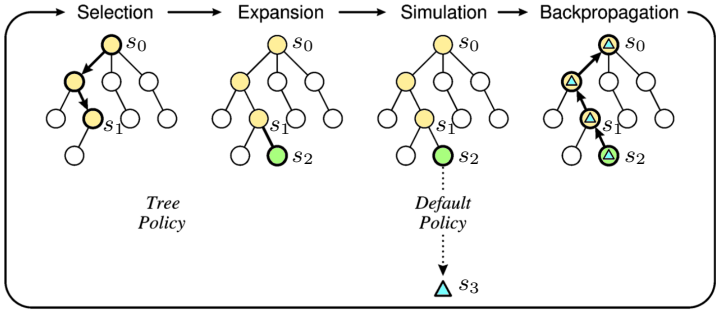

Fig. 2. The MCTS algorithm iteratively builds a subtree of the possible future states (circles). This figure (from [4]) illustrates one iteration of the algorithm. Starting from the root node s_0 (current state of the problem), a node s_1 is selected and a new node s_2 is created. A playout is performed (until a final state s_3 is reached) and the subtree is updated.

where $\mathcal{C}_{s_1}$ is the set of child nodes of the node s_1, w_j is the number of wins for the node j (more precisely, the sum of the final rewards for j), n_j is the number of playouts for the node j and n_{s_1} is the number of playouts for the node s_1 ($n_{s_1} = \sum_j n_j$). K is called the exploration parameter and is used to tune the trade-off between exploitation and exploration.

Once a leaf node s_1 is selected, the *expansion* step creates a new child node s_2. This new node corresponds to a decision of s_1 which has not been considered yet. Then, the *simulation* step is to perform a playout (a random game) until a final state s_3 is reached. This final state gives a reward (for example, in games, the reward corresponds to a win, a loss or a draw). The last step (*backpropagation*) is to use the reward to update the statistics (number of wins and number of playouts) in all the nodes encountered during the *selection* step.

3.1 Rapid Action Value Estimate

One of the most common improvements of the MCTS algorithm is the Rapid Action Value Estimate (RAVE) [12]. The idea is to share some statistics about moves between nodes: if a move is good in a certain state, then it may be good in other ones.

More precisely, let s be a node and m_i the possible moves from s, leading to the child nodes s'_i. For the classic MCTS algorithm, we already store, in s, the number of winning playouts w_s and the total number of playouts n_s (after s was selected). For the RAVE improvement, we also store, in s and for each move m_i, the number of winning playouts w'_{s,s'_i} and the total number of playouts n'_{s,s'_i} obtained by choosing the move m_i. These "RAVE statistics" are updated during the backpropagation step and indicate the estimated quality of the moves already considered in the subtree (see Fig. 3).

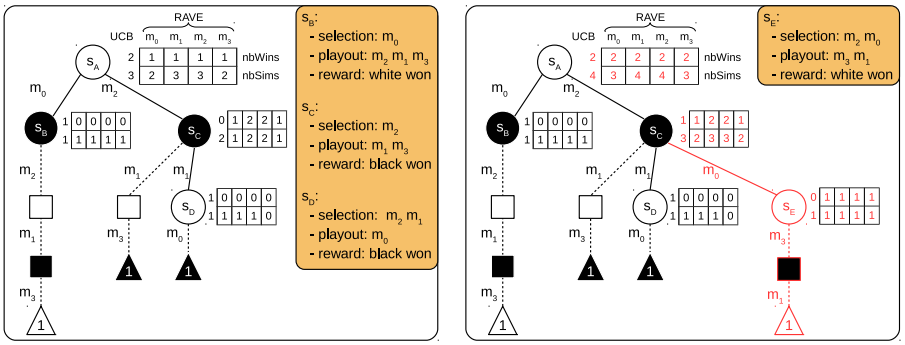

Fig. 3. Illustration of the RAVE process. In each node, an array stores the RAVE statistics of all possible moves (left); this array is updated when a corresponding move is played (right). In this example, a new node (S_E) is created and all the moves chosen in the *selection* step (m_2,m_0) and in the *simulation* step (m_3,m_1) are updated in the RAVE statistics of the selected nodes (S_A,S_C,S_E) during the *backpropagation* step.

Thus, the selection step can be biased by adding a RAVE score in the bandit formula defined previously:

$$s_1 \leftarrow \underset{j \in \mathcal{C}_{s_1}}{\arg\max} \left[(1-\beta)\ \frac{w_j}{n_j} + \beta\ \frac{w'_{s_1,j}}{n'_{s_1,j}} + K\ \sqrt{\frac{\ln(n_{s_1})}{n_j}} \right],$$

where β is a parameter approaching 0 as n_j tends to infinity (for instance, $\beta = \sqrt{\frac{R}{R+3n_j}}$ where R is a parameter [13]).

3.2 Playout improvements

PoolRave is an extension of RAVE [25, 17]. The idea is to use the RAVE statistics to bias the simulation step (unlike the RAVE improvement which biases the selection step). More precisely, when a playout is performed, the PoolRave improvement firstly builds a pool of possible moves by selecting the N best moves according to the RAVE statistics. Then, in the simulation step, the moves are chosen randomly in the pool with probability p, otherwise (with probability $1-p$) a random possible move is played, as in the classic MCTS algorithm.

The Last-Good-Reply improvement [8, 2] is based on the principle of learning how to respond to a move. In each node, LGR stores move replies which lead to a win in previous playouts. More precisely, during a playout, if the node has a reply for the last move of the opponent, this reply is played, otherwise a new reply is created using a random possible move. At the end of the playout, if the playout leads to a win, the corresponding replies are stored in the node. If the playout leads to a loss, the corresponding replies are removed from the node (*forgetting* step). This algorithm is called LGRF1. Other algorithms have been proposed

using the same idea but LGRF1 is the most efficient one with connection games [27].

The principle of the Move-Average Sampling Technique (MAST) [11] is to store move statistics globally and to use these statistics to bias the playouts. This is similar to the PoolRave improvement, except that here, the statistics are independent of the position of the move in the tree.

The N-gram Average Sampling Technique (NAST) is a generalization of MAST [23, 28]. The idea is to look at sequences of N moves instead of one move only. This improvement can be costly according to N but it is already efficient with $N = 2$ (NAST2) for the game of Havannah [27].

4 Pruning in the simulation step

We propose a new improvement of the MCTS algorithm, called "Playout Pruning with Rave" (PPR). The idea is to prune bad moves in the simulation step in order to focus the simulation on good playouts (see Fig. 4, left). More precisely, before the playout, we compute a list of good moves by pruning the moves which have a winning rate lower than a given threshold $T_{w'}$. The winning rate of a node j is computed using the RAVE statistics of a node s_{PPR}, with $\frac{w'_{s_{\text{PPR}},j}}{n'_{s_{\text{PPR}},j}}$.

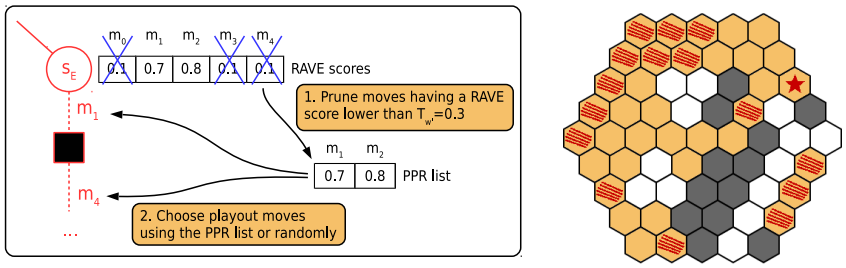

Fig. 4. During a playout (left), the PPR process discards all moves with a RAVE winning rate lower than a given threshold, then plays a move among this pruned list (or a random move, according to a given probability). For example (right), after 100k MCTS iterations for black, PPR prunes the scratched cells and finally plays the starred cell, which seems relevant: the three scratched cells on the right cannot be used by black to form a winning shape; at the top left of the board several white cells prevent black from accessing the scratched cells easily; the three remaining scratched cells are seen by PPR as functionally equivalent to other possible cells of the board.

The node s_{PPR}, giving the RAVE statistics, has to be chosen carefully. Indeed, the node s_2, selected during the selection step of the MCTS algorithm, may still have very few playouts, hence inaccurate RAVE statistics. To solve this problem, we traverse the MCTS tree bottom-up, starting from s_2, until we reach a node

with a minimum ratio T_n, representing the current number of playouts for s_{PPR} over the total number of playouts performed.

After the PPR list is computed, the simulation step is performed. The idea is to use the moves in the PPR list, which are believed to be good, but we also have to choose other moves to explore other possible playouts. To this end, during the simulation step, each move is chosen in the PPR list with a probability p, or among the possible moves with a probability $1-p$. In the latter case, we have observed that considering only a part of all the possible moves gives better results; this can be seen as a default pruning with, in return, an additional bias (see Algorithm 1).

Algorithm 1 : Monte-Carlo Tree Search with RAVE and PPR

{initialization}
$s_0 \leftarrow$ create root node from the current state of the problem

while there is some time left **do**

{selection}
$s_1 \leftarrow s_0$
while all possible decisions of s_1 have been considered **do**
$C_{s_1} \leftarrow$ child nodes of s_1
$\beta \leftarrow \sqrt{\frac{R}{R+3n_j}}$
$s_1 \leftarrow \underset{j \in C_{s_1}}{\arg\max} \left[(1-\beta)\frac{w_j}{n_j} + \beta \frac{w'_{s_1,j}}{n'_{s_1,j}} + K\sqrt{\frac{\ln(n_{s_1})}{n_j}} \right]$

{expansion}
$s_2 \leftarrow$ create a child node of s_1 from a possible decision of s_1 not yet considered

{pruning}
$s_{\mathrm{PPR}} \leftarrow s_2$
while $n_{s_{\mathrm{PPR}}} < T_n$ **do**
$s_{\mathrm{PPR}} \leftarrow$ parent node of s_{PPR}
$\mathrm{PPR} \leftarrow \{\ j \mid \frac{w'_{s_{\mathrm{PPR}},j}}{n'_{s_{\mathrm{PPR}},j}} > T_{w'}\ \}$

{simulation/playout}
$s_3 \leftarrow s_2$
while s_3 is not a terminal state for the problem **do**
$\xi \leftarrow$ random()
if $\xi \leq p$ **then**
$s_3 \leftarrow$ randomly choose next state in PPR
else
$s_3 \leftarrow$ randomly choose next state in the $(1-\xi)$ last part of the possible moves

{backpropagation}
$s_4 \leftarrow s_2$
while $s_4 \neq s_0$ **do**
$w_{s_4} \leftarrow w_{s_4} +$ reward of the terminal state s_3 for the player of s_4
$n_{s_4} \leftarrow n_{s_4} + 1$
for all nodes j belonging to the path $s_0 s_3$ **do**
$w'_{s_4,j} \leftarrow w'_{s_4,j} +$ reward of the terminal state s_3 for the player of j
$n'_{s_4,j} \leftarrow n'_{s_4,j} + 1$
$s_4 \leftarrow$ parent node of s_4

return best child of s_0

The PPR improvement can be seen as a dynamic version of the PoolRave improvement presented in the previous section: instead of selecting the N best moves in a pool, we discard the moves which have a winning rate lower than $T_{w'}$. PoolRave uses a static pool size, which implies that good moves may be discarded (if the pool size is small in front of the number of good moves) or that bad moves may be chosen (if the pool size is large in front of the number of good moves). PPR automatically deals with this problem since the size of the PPR list is naturally dynamic: the list is small if there are only few good moves, and large if there are many good moves.

5 Experiments

We have experimented with the proposed MCTS improvement (PPR) for the game of Havannah. Since RAVE is now considered as a classic MCTS baseline, we have compared PPR against RAVE (using the parameters $R = 130$ and $K = 0$). To have good statistical properties, we have performed 600 games for each experiment. Since the first player has an advantage in the game of Havannah, we perform, for each experiment, half the games with the first algorithm as the first player and the other half with the second algorithm as the first player.

5.1 Influence of the PPR parameters

To study the influence of the three parameters of the PPR improvement, we have compared PPR against RAVE using 1k MCTS iterations and a board size of 6. For each parameter, we have experimented with various values while the other parameters were set to default values (see Fig. 5).

PPR has better win rates against RAVE when T_n (the minimum ratio of playouts for the node s_{PPR} over the total number of playouts) is lower than 10%. A low value for T_n means that we take a node s_{PPR} close to the node s_2 which has launched the playout; thus the PPR list is built using RAVE statistics that are meaningful for the playout but quite unreliable. When T_n is too large, no node has enough playouts so the PPR list is empty and PPR is equivalent to RAVE (win rate of 50%).

The best values for the pruning threshold $T_{w'}$ (win rate in the RAVE statistics of s_{PPR}) stand between 20% and 40%. The moves with a winning rate lower than this threshold are pruned when building the PPR list. Therefore, if $T_{w'}$ is too high, all moves are pruned (i.e. the PPR list is empty) and the algorithm is equivalent to RAVE (win rate of 50%). On the other hand, if $T_{w'}$ is too low, then the PPR list also contains bad moves (low winning rate) which lowers the efficiency of PPR.

Finally, the best values for the parameter p (probability for using the PPR list instead of a random sampling, to choose a move) stand between 60% and 80% in our experiments. A low value implies that the PPR list is rarely used, making PPR almost equivalent to RAVE. With a very high value, the PPR list is frequently used, so PPR does not explore other moves, hence a highly biased playout computation.

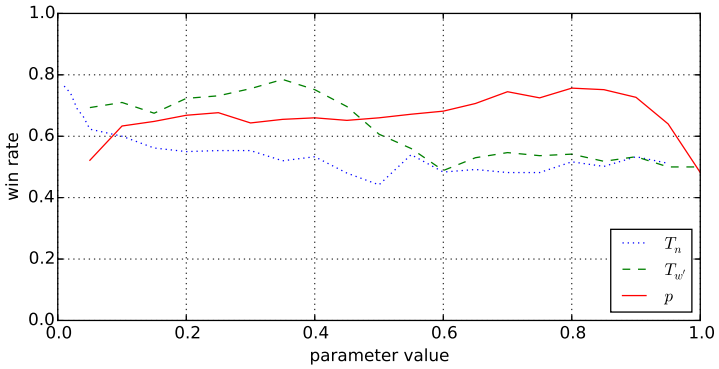

Fig. 5. Influence of the PPR parameters in the game of Havannah (PPR vs RAVE, 1k MCTS iterations, board size 6). Each parameter is studied while the other ones are set to default values: $T_n = 1\%$, $T_{w'} = 25\%$ and $p = 80\%$, where T_n is the minimum ratio of playouts for the node s_{PPR}, $T_{w'}$ is the win rate threshold for pruning bad moves and p is the probability for using the PPR list.

5.2 Scalability of the playout pruning

Like classic improvements of the simulation step (for instance, PoolRave and LGRF1), PPR is useful for small numbers of playouts and large board sizes (see Fig.6).

In our experiments, PPR wins almost 80% of the games against RAVE with 1k MCTS iterations, and almost 70% with 10k iterations. PPR wins 60% or less of the games against RAVE with a board size lower than 5 and 80% or more of the games with a board size larger than 7. This is not very surprising because RAVE is already very efficient when the board size is small, so adding pruning is useless in this case. However, large boards have a lot more "dead areas" (i.e. irrelevant cells) that PPR can detect and prune (see Fig. 4, right).

5.3 PPR vs other playout improvements

We have compared PPR against several MCTS improvements (RAVE, PoolRave, LGRF1, MAST, NAST2) for several board sizes and numbers of MCTS iterations (see Table 1). Since RAVE is now considered as the classic MCTS baseline, we have implemented all playout improvements (PPR, PoolRave, LGRF1, MAST, NAST2) based on the RAVE algorithm.

Our results indicate that PPR outperforms the previous algorithms for the game of Havannah. For a board size of 6, PPR wins more than 70% of the games with 1k MCTS iterations and more than 60% of the games with 10k or 30k iterations. For a board size of 10, PPR is even better (more than 70%).

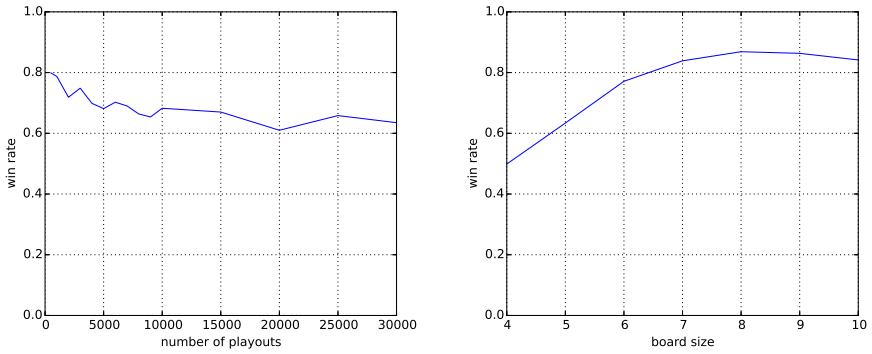

Fig. 6. Influence of the number of MCTS iterations (left, with board size 6) and board size (right, with 1k MCTS iterations) in the game of Havannah (PPR vs RAVE, $T_n = 1\%$, $T_{w'} = 25\%$ and $p = 80\%$).

Table 1. PPR vs other MCTS improvements. We have performed 200 games for the experiments with size=10 and playouts=30,000; 600 games for the other experiments.

size	playouts	player	win rate	std dev
6	1,000	Rave	74.4%	±1.78
		PoolRave	70.17%	±1.87
		LGRF1	71.67%	±1.84
		MAST	74.0%	±1.79
		NAST2	85.0%	±1.46
	10,000	Rave	63.67%	±1.96
		PoolRave	67.0%	±1.92
		LGRF1	63.17%	±1.97
		MAST	64.5%	±1.95
		NAST2	76.5%	±1.73
	30,000	Rave	66.33%	±1.92
		PoolRave	73.66%	±1.79
		LGRF1	65.66%	±1.93
		MAST	65.5%	±1.94
		NAST2	60.5%	±1.99

size	playouts	player	win rate	std dev
10	1,000	Rave	86.33%	±1.40
		PoolRave	72.16%	±1.82
		LGRF1	79.00%	±1.66
		MAST	83.66%	±1.50
		NAST2	85.50%	±1.43
	10,000	Rave	79.16%	±1.65
		PoolRave	89.00%	±1.27
		LGRF1	83.83%	±1.50
		MAST	79.00%	±1.66
		NAST2	85.16%	±1.45
	30,000	Rave	75.85%	±2.13
		PoolRave	91.01%	±1.42
		LGRF1	79.69%	±2.01
		MAST	82.04%	±1.91
		NAST2	84.08%	±1.82

6 Conclusion

In this paper, we have proposed a new improvement (called PPR) of the MCTS algorithm, based on the RAVE improvement. The idea is to prune the moves which seem "bad" according to previous playouts during the simulation step. We have compared PPR to previous MCTS improvements (RAVE, PoolRave, LGRF1, MAST, NAST2) for the game of Havannah. In our experiments, PPR is the most efficient algorithm, reaching win rates of at least 60%.

In future work, it would be interesting to compare PPR with other MCTS improvements such as Contextual Monte-Carlo [24] or with stronger bots [10]. We would also try PPR for other games or decision making problems to determine if the benefit of PPR is limited to the game of Havannah or if it is more general.

Acknowledgements

Experiments presented in this paper were carried out using the CALCULCO computing platform, supported by SCOSI/ULCO (Service Commun du Système d'Information de l'Université du Littoral Côte d'Opale).

References

1. Arneson, B., Hayward, R., Henderson, P.: Monte-Carlo Tree Search in Hex. Computational Intelligence and AI in Games, IEEE Transactions on 2(4), 251–258 (2010)
2. Baier, H., Drake, P.: The power of forgetting: Improving the last-good-reply policy in Monte-Carlo Go. Computational Intelligence and AI in Games, IEEE Transactions on 2(4), 303–309 (Dec 2010)
3. Bertsimas, D., Griffith, J., Gupta, V., Kochenderfer, M.J., Mišić, V., Moss, R.: A comparison of Monte-Carlo Tree Search and Mathematical optimization for large scale dynamic resource allocation. arXiv:1405.5498 (2014)
4. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte-Carlo Tree Search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1), 1–43 (2012)
5. Cazenave, T.: Monte-Carlo Kakuro. In: Advances in Computer Games. Lecture Notes in Computer Science, vol. 6048, pp. 45–54. Springer (2009)
6. Chaslot, G., Saito, J., Bouzy, B., Uiterwijk, J., Herik, H.J.V.D.: Monte-Carlo strategies for computer Go. In: Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium. pp. 83–91 (2006)
7. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo Tree Search. In: Computers and games, pp. 72–83. Springer (2007)
8. Drake, P.: The last-good-reply policy for Monte-Carlo Go. International Computer Games Association Journal 32(4), 221–227 (2009)
9. Edelkamp, S., Tang, Z.: Monte-Carlo Tree Search for the multiple sequence alignment problem. In: Eighth Annual Symposium on Combinatorial Search (2015)
10. Ewalds, T.: Playing and Solving Havannah. Master's thesis, University of Alberta (2012)

11. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1. pp. 259–264. AAAI'08, AAAI Press (2008)
12. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Proceedings of the 24th international conference on Machine learning. pp. 273–280. ACM (2007)
13. Gelly, S., Silver, D.: Monte-Carlo Tree Search and rapid action value estimation in computer Go. Artificial Intelligence 175(11), 1856–1875 (2011)
14. Guo, X., Singh, S., Lee, H., Lewis, R.L., Wang, X.: Deep learning for real-time atari game play using offline Monte-Carlo Tree Search planning. In: Advances in Neural Information Processing Systems. pp. 3338–3346 (2014)
15. Heinrich, J., Silver, D.: Self-play Monte-Carlo Tree Search in computer Poker. In: Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence (2014)
16. Herik, H.J.V.D., Kuipers, J., Vermaseren, J., Plaat, A.: Investigations with Monte-Carlo Tree Search for finding better multivariate horner schemes. In: Agents and Artificial Intelligence, pp. 3–20. Springer (2014)
17. Hoock, J., Lee, C., Rimmel, A., Teytaud, F., Wang, M., Teytaud, O.: Intelligent agents for the game of Go. Computational Intelligence Magazine, IEEE 5(4), 28–42 (2010)
18. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Machine Learning: ECML 2006. pp. 282–293. Springer (2006)
19. Lanctot, M., Saffidine, A., Veness, J., Archibald, C., Winands, M.: Monte Carlo*-minimax search. In: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence. pp. 580–586. AAAI Press (2013)
20. Lorentz, R.: Improving Monte-Carlo Tree Search in Havannah. In: Computers and Games'10. pp. 105–115 (2010)
21. Lorentz, R.: Amazons discover Monte-Carlo. In: Computers and games, pp. 13–24. Springer (2008)
22. Mazyad, A., Teytaud, F., Fonlupt, C.: Monte-Carlo Tree Search for the "mr jack" board game. Journal on Soft Computing, Artificial Intelligence and Applications (IJSCAI) 4(1) (2015)
23. Powley, E.J., Whitehouse, D., Cowling, P.I.: Bandits all the way down: UCB1 as a simulation policy in Monte-Carlo Tree Search. In: CIG. pp. 81–88. IEEE (2013)
24. Rimmel, A., Teytaud, F.: Multiple overlapping tiles for contextual Monte-Carlo Tree Search. Applications of Evolutionary Computation pp. 201–210 (2010)
25. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo simulations through rave values. Computers and Games pp. 59–68 (2011)
26. Schmittberger, R.: New Rules for Classic Games. Wiley (1992)
27. Stankiewicz, J., Winands, M., Uiterwijk, J.: Monte-Carlo Tree Search enhancements for Havannah. In: Advances in Computer Games, pp. 60–71. Springer (2012)
28. Tak, M.J., Winands, M.H., Björnsson, Y.: N-grams and the last-good-reply policy applied in general game playing. Computational Intelligence and AI in Games, IEEE Transactions on 4(2), 73–83 (2012)
29. Taralla, D.: Learning Artificial Intelligence in Large-Scale Video Games. Ph.D. thesis, University of Liège (2015)
30. Teytaud, F., Teytaud, O.: Creating an upper-confidence-tree program for Havannah. Advances in Computer Games pp. 65–74 (2010)
31. Wilisowski, L., Dreżewski, R.: The application of co-evolutionary genetic programming and td (1) reinforcement learning in large-scale strategy game vcmi. In: Agent and Multi-Agent Systems: Technologies and Applications, pp. 81–93. Springer (2015)