On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms

Fabien Teytaud, Olivier Teytaud

On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms

Fabien Teytaud, Olivier Teytaud
TAO (Inria), LRI, UMR 8623(CNRS - Univ. Paris-Sud),
bat 490 Univ. Paris-Sud 91405 Orsay, France

***Abstract*— Monte-Carlo Tree Search (MCTS) algorithms, including upper confidence Bounds (UCT), have very good results in the most difficult board games, in particular the game of Go. More recently these methods have been successfully introduce in the games of Hex and Havannah. In this paper we will define decisive and anti-decisive moves and show their low computational overhead and high efficiency in MCTS.**

I. Introduction

MCTS[10], [8] and UCT[18] are now well established as strong candidates for planning and games, in particular when (i) the dimensionality is huge (ii) there's no efficient handcrafted value function. They provided impressive results in the game of Go[20], in connection games[6], [29], and in the important problem of general game playing[28]; this suggests the strong relevance of MCTS and UCT for general purpose game tools. These techniques were also applied to Partially Observable Markov Decision Processes derived from fundamental artificial intelligence tasks [25], [3] that were unsolvable by classical Bellman-based algorithms, and related techniques also provided some world records in one-player games[26], [5]. An industrial application was successful in a difficult context in which the baseline was heavily optimized[14].

A complete introduction to UCT and MCTS is beyond the scope of this paper; we essentially recall that:

- following [4], MCTS is a Monte-Carlo approach, i.e. it is based on many random games simulated from the current board;
- following [10], [8], these random games are progressively biased toward better simulations; this bias follows a "bandit" formula, which can be Upper Confidence Bounds[19], [2], [30], [1] (like in UCT[18]), or more specialized formulas[20] combining offline learning and online estimates; the general idea is presented in Alg. 1 for the UCT version. Bandits formulas are formulas aimed at a compromise between exploration (analyzing moves which have not yet been sufficiently analyzed) and exploitation (analyzing moves which are better and therefore more likely).
- when the time is elapsed, then the most simulated move from the current board is chosen.

This simple algorithm is anytime[33] in the sense that it is reasonably efficient whatever may be the computational power per move, with better and better results as long as the computational power per move increases. It outperforms many algorithms and in particular the classical $\alpha - \beta$ approach in many difficult games. It can be implemented provided that:

- the rules are given (function nextState$(state, move)$);
- a default policy is given (a default solution is the use of a pure uniform random move).

A main advantage is therefore that we don't need/don't use an evaluation function. This is obviously a trouble when very efficient value functions exist, but it is great for games in which no good value function exist. Also, we can use offline learning for improving the bandit formula 1, or rapid-action-value-estimates[16]. When an evaluation function is available, it can be used by cutting the Monte-Carlo part

Algorithm 1 The UCT algorithm in short. $nextState(s, m)$ is the implementation of the rules of the game, and the ChooseMove() function is defined in Alg. 2. The constant k is to be tuned empirically.

```
d =UCT(situation s_0, time t)
while Time left > 0 do
  s = s_0    // start of a simulation
  while s is not terminal do
    m =ChooseMove(s)
    s =nextState(s, m)
  end while
    // the simulation is over
end while
d = most simulated move from s_0 in simulations above
```

Algorithm 2 The ChooseMove function, which chooses a move in the simulations of UCT.

ChooseMove(situation s)
if There's no statistics from previous simulations in s **then**
  Return a move randomly according to some default policy
else
  for Each possible move m in s **do**
    compute a score(m) as follows:

$$\text{average reward when choosing } m \text{ in } s + \sqrt{\frac{k.\log(\text{nb of simulations in } s)}{\text{nb of simulations of } m \text{ in } s}}. \quad (1)$$

  end for
  Return the move with highest score.
end if

and replacing the end of it by the evaluation function; this promising technique has shown great results in [21].

However, a main weakness is the handcrafting in the Monte-Carlo part. The default uniform policy is not satisfactory and strong results in the famous case of the game of Go appeared when a good Monte-Carlo was proposed. How to derive a Monte-Carlo part, i.e. a default policy ? It is known that playing well is by no mean the goal of the default policy: one can derive a much stronger default policy than those used in the current best implementations, but the MCTS algorithm built on top of that is indeed much weaker!

The use of complex features with coefficients derived from databases was proposed in [11], [9] for the most classical benchmark of the game of Go, but the (tedious) handcrafting from [31] was adopted in all efficient implementations, leading to sequence-like simulations in all strong implementations for the game of Go. Also, some very small modifications in the Monte-Carlo part can have big effects, as the so-called "fillboard" modification proposed in [7] which provides a 78 % success rate against the baseline with a not-so-clear one-line modification. A nice solution for increasing the efficiency of the Monte-Carlo part is nested Monte-Carlo[5]; however, this technique has not given positive results in two-player games. We here propose a simple and fast modification, namely decisive moves, which strongly improves the results.

Connection games (Fig. 3) are an important family of board games; using visual features, humans are very strong opponents for computers. Connection games like Hex (invented by John Nash as a development around the game of Go), Y (invented by Claude Shannon as an extension of Hex), Havannah (discussed below), TwixT, *Star have very simple rules and huge complexity; they provide nice frameworks for complexity analysis:

- no-draw property, for Hex [22] and probably Y (a first flawed proof was published and corrected);
- first player wins in case of perfect play, for Hex, when no pie-rule is applied (otherwise, the second player wins). This is proved by the strategy-stealing argument: if the second player could force a win, then the first player could adapt this winning strategy to his case and win first.
- Many connection games are PSPACE as they proceed by adding stones which can never been removed and are therefore solved in polynomial space by a simple depth-first-search (Hex, Havannah, Y, *Star and the versions of Go with bounded numbers of captures like Ponnuki-Go); most of them are indeed proved or conjectured also PSPACE-hard (they're therefore proved or conjectured PSPACE-complete); an important result is the PSPACE-completeness of Hex[23].

They are also classical benchmarks for exact solving (Jing Yang solved Hex for size 9x9) and artificial intelligence[29], [6]. The case of Go is a bit different as there are captures; some variants are known EXPTIME-complete[24], some other PSPACE-hard[13], and some cases are still unknown; as there are plenty of families of situations in Go, some restricted cases are shown NP-complete as well [12]. In all the paper, $[a,b] = \{x; a \leq x \leq b\}$ and $[[a,b]] = \{0,1,2,\dots\} \cap [a,b]$. $\log*(n)$ (sometimes termed the iterated logarithm) is defined as $\log*(1) = 0$, $\log*(2) = 1$ and $\log*(n) = 1 + \log*(\log(n)/\log(2))$; $\log*(n)$ increases very slowly to ∞ and in particular $\log*(n) = o(\log(n))$, so that complexities in $T\log(T)$ are bigger than complexities in $T\log*(T)$.

Section II introduces the notion of decisive moves (i.e. moves which conclude the game), and anti-decisive moves (i.e. moves which avoid decisive moves by the opponent one step later). Then section III then presents connection games, in the general case (section III-A), in the case of Hex and Havannah (section III-B), and then the data structure (section III-C) and a complexity analysis of decisive moves in this context (section III-D). Section IV presents experiments.

II. Decisive moves and anti-decisive moves

In this section, we present decisive moves and their computational cost. When a default policy is available for some game, then it can be rewritten as a version with decisive moves as follows:

- If there is a move which leads to an immediate win, then play this move.
- Otherwise, play a move (usually randomly) chosen by the default (Monte-Carlo) policy.

This means that the ChooseMove function from the UCT algorithm (see Alg. 2) is modified as shown in Alg. 3. Yet another modification, termed "anti-decisive moves", is presented in Alg. 4.

Algorithm 3 ChooseMove() function, with decisive moves. To be compared with the baseline version in Alg. 2.

ChooseMove$_{DM}$**(situation** s**)**
  // version with decisive moves
if There is a winning move **then**
  Return a winning move
else
  Return $ChooseMove(s)$
end if

This can be related to the classical quiescent search[17] in the sense that it avoids to launch an evaluation (here a Monte-Carlo evaluation) in an unstable situation.

III. Connection games, and the case of Havannah

In order to have a maximal generality, we first give the general conditions under which our complexity analysis applies. We will then make things more concrete by considering Hex and Havannah; the section below can be skipped in first reading.

Algorithm 4 ChooseMove() function, with decisive moves and anti-decisive moves. To be compared with the baseline version in Alg. 2 and the version with decisive moves only in Alg. 3.

ChooseMove$_{DM+ADM}$**(situation** s**)**
  // version with decisive and
  // antidecisive moves
if There is a winning move **then**
  Return a winning move
else
  if My opponent has a winning move **then**
    Return a winning move of my opponent
  else
    Return $ChooseMove(s)$
  end if
end if

A. Connection games

We will consider here the complexity of decisive moves in an abstract framework of connection games; more concrete examples (Hex and Havannah) are given in the section and the current section can therefore be skipped without trouble by people who want to focus on some clear examples first. We consider the complexity for a machine which contains integer registers of arbitrary size, and random access to memory ($O(1)$ cost independently of the location of the memory part). Games under consideration here are as follows for some data structure representing the board:

- The game has (user chosen) size T in the sense that there are T locations and at most T time steps.
- The data structure d_t at time step t contains
  - the current state $d_t.s$ of the board;
  - for each location l, some information $d_t.s(l)$ which is sufficient for checking in $O(1)$ whether playing in l is an immediate win for the player to play at situation $d_t.s$;
  - for each location l, the list of $d_t.s(l).n_{timeSteps}$, supposed to be at most $O(\log(t))$ time steps at which the local information $d_t.s(l)$ has changed;
  - for each location l and each time step $u \in [[1, d_t.s(l).n_{timeSteps}]]$, the local information at time step u, i.e. $d_u.s(l)$.
- "Monotonic" games: one more stone for a given player at any given location can change the situation into a win, but can never replace a win by a loss or a draw; when a stone is put on the board, it is never removed.
- The update of the data structure is made in time $O(T \log(T))$ for a whole game, for any sequence of moves.

We also assume that the Monte-Carlo choice of a move can be made in time $O(1)$ for the default (Monte-Carlo) policy; this is clearly the case

- with the uniform Monte-Carlo thanks to a list of empty locations;
- for local pattern-matching like in [31].

We also assume that the initial state can be built in memory in time $O(T)$. As this is somehow abstract, we will give a concrete example for Hex and/or Havannah in next section.

B. Rules of Hex and Havannah

The rules of Hex are very simple: there's a rhombus (with sides A,B,C,D) (Fig. 3) equipped with an hexagonal grid; each player, in turns, fill an empty location with a stone of his color. Player 1 wins if he connects sides A and C, player 2 wins if he connects sides B and D. The game of Havannah is a board game, recently created by Christian Freeling [27], [32]. In this game, two players (black and white) put in an empty location of the board. This is an hexagonal board of hexagonal locations, with variable sizes (most popular sizes for humans are 8 or 10 hexes per side).
The rules are very simple:

- White player starts.
- Each player put a stone on one empty cell. If there is no empty cell and if no player has won yet, then the game is a draw (this almost never happens).
- To win the game, a player has to realize one of the three following structures:
  - A ring, which is a loop around one or more cells. These surrounded cells can be black or white stones, or empty cells.
  - A bridge, which is a continuous string of stones connected to two of the six corners.
  - A fork, which is a continuous string of stones connected to three of the six sides. Corner cells do not belong to a side.

These three winning positions are presented in Fig. 1.

For classic board sizes, the best computers in Havannah are weak compared to human experts. To show that, in 2002, Christian Freeling, the game inventor, offered a prize of 1000 euros, available through 2012, for any computer program that could beat him one game of a ten game match.
The main difficulties of this game are :

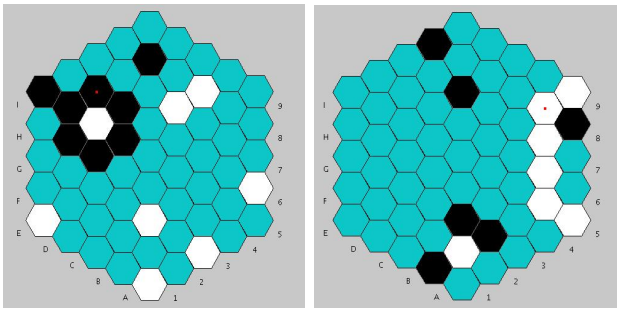
- There's almost no easy to implement expert knowledge;
- No natural evaluation function;
- No pruning rule for reducing the number of possible moves;
- Large action space. For instance, the first player, on an empty board of size 10 has 271 possible moves.

MCTS has been recently introduced for the game of Havannah in [29].

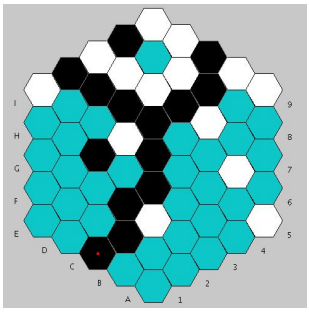
C. Data structures for Hex and Havannah

The data structure and complexities above can be realized for Hex and Havannah (and more generally for many games corresponding to the intuitive notion of connection games) as follows:

- For each location l, we keep as information in the state d_t for time step t the following $d_t.s(l)$:
  - the color of the stone on this location, if any;



Ring (black) Bridge (white)



Fork (black)

Fig. 1. Three finished Havannah games: a ring (a loop, by black), a bridge (linking two corners, by white) and a fork (linking three edges, by black).

  - if there is a stone, a group number; connected stones have the same group number;
  - the time steps $u \leq t$ at which the local information $d_u.s(l)$ has changed; we will see below why this list has size $O(\log(T))$; the group information and the connections for all stones in the neighborhood of l are kept in memory for each of these time steps.
- For each group number, we keep as information:
  - the list of edges/corners to which this group is connected (in Hex, corners are useless, and only some edges are necessary);
  - the number of stones in the group;
  - one location of a stone in this group.
- For each group number and each edge/corner,
  - the time step (if any) at which this group was connected to this edge.
- At each move, all the information above is updated. The important thing for having $O(T\log(T))$ overall complexity in the update is the following: when k groups are connected, then the local information should be changed for the $k-1$ smallest groups and not for the biggest group (see Fig. 2). This implies that each local information is updated at most $O(\log(T))$ times because the size of the group, in case of local update, is at least multiplied by k.

Checking a win in $O(1)$ is easy by checking connections of groups modified by the current move (for fork and bridge) and by checking local information for cycles:

- a win by fork occurs if the new group is connected to 3 edges;
- a win by bridge occurs if the new group is connected to 2 corners;
- a win by cycle occurs when a stone connects two stones of the same group, at least under some local conditions which are tedious but fast to check locally.

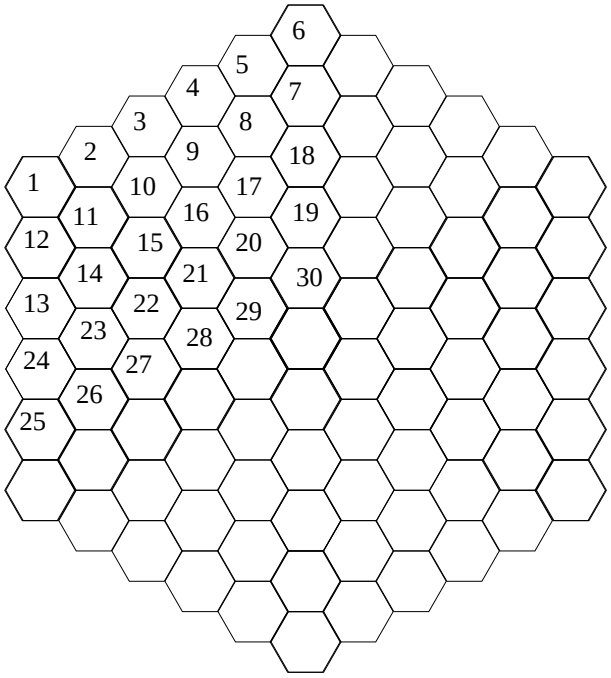


Fig. 2. An example of sequence of $\Theta(T)$ moves for one of the players which cost highly depends on the detailed implementation. The important feature of this sequence of moves is that (i) it can be extended to any size of board (ii) the size of the group increases by 1 at each move of the player. In this case, if the biggest group has its information updated at each new connection, then the total cost is $\Theta(T^2)$; whereas if the smallest group is updated, the total cost is $\Theta(T)$ for this sequence of moves, and $\Theta(T\log(T))$ in all cases. Randomly choosing between modifying the 1-stone group and the big group has the same expected cost $\Theta(T^2)$ as the choice of always modifying the big group (up to a factor of 2).

D. Complexity analysis

Under conditions above, we can:

- initialize the state ($O(T)$);
- T times,
  - randomly choose a move (cumulated cost $O(T)$);
  - update the state (cumulated cost $O(T\log(T))$);
  - check if this is a win (cumulated cost $O(T)$ - exit the loop in this case).

therefore we can perform one random simulation in time $O(T\log(T))$. This is not optimal, as union/find algorithms [15] can reach $O(T\log*(T))$; but the strength of this data structure is that we can switch to decisive moves with no additional cost (except within a constant factor). This is performed as follows:

- initialize the state ($O(T)$);
- T times,
  - randomly choose a move (cumulated cost $O(T)$);
  - update the state (cumulated cost $O(T\log(T))$);

  - check if this is a win (cumulated cost $O(T)$) (exit the loop in this case).
- let $firstWin$ =time step at which the above game was won ($+\infty$ in case of draw).
- let $winner$ =the player who has won.
- for each time location l, ($O(T)$ times)
  - for each time step t (there are at most $O(\log(T))$ such time steps by assumption on the data structure) at which $d_t.s(l)$ has changed, ($O(\log(T))$ times)
    * check if $d_t.s(l)$ was legal and a win for the player p to play at time step t; ($O(1)$)
    * if yes, if $t < firstWin$ then $winner = p$ and $firstWin = t$; ($O(1)$)
    * check if $d_{t+1}.s(l)$ was legal and a win for the player p to play at time step $t+1$; ($O(1)$)
    * if yes, if $t+1 < firstWin$ then $winner = p$ and $firstWin = t+1$. ($O(1)$)

The overall cost is $O(T \log(T))$. We point out the following elements:

- The algorithm above consists in playing a complete game with the default policy, and then, check if it was possible to win earlier for one of the players. This is sufficient for the proof, but maybe it is much faster (at least from a constant) to check this during the simulation.
- We do not prove that it's not possible to reach $T \log *(T)$ with decisive moves in Hex or Havannah; just, we have not found better than $T \log(T)$.

IV. Experiments

We perform our experiments on Havannah; the game of Havannah is a connection game with a lot of interest from the computer science community (see [27], [29], `littlegolem.net` and `boardgamegeek.net`). It involves more tricky elements than Hex and it is therefore a good proof of concept. Please note that we do not implemented the complete data structure above in our implementation, but some simpler tools which are slower but have the advantage of covering anti-decisive moves as well. We have no proof of complexity for our implementation and no proof that the $T \log(T)$ can be reached for anti-decisive moves.

We implement the decisive moves and anti-decisive moves in our Havannah MCTS-bot for measuring the corresponding improvement. We can see in Table I that adding decisive move can lead to big improvements; the modification scales well in the sense that it becomes more and more effective as the number of simulations per move increases.

V. Discussion

We have shown that (i) decisive moves have a small computational overhead ($T \log(T)$ instead of $T \log *(T)$) (ii) they provide a big improvement in efficiency. The improvement increases as the computational power increases.

Anti-decisive moves might have a bigger overhead, but they are nonetheless very efficient as well even with fixed time per move. A main lesson, consistent with previous works in Go, is that having simulations with a better scaling as a function of the computational power, is usually a good idea whenever these simulations are more expensive.

TABLE I

Success rates of decisive moves. BL is the baseline (no decisive moves). DM corresponds to BL plus the "decisive moves" (if there exists a winning move then it is played). DM + ADM, is the DM version of our bot, plus the "antidecisive moves" improvement: in that case, if player p is to play, if p has a winning move m then p plays m; else, if the opponent has a winning move m', then p plays m'.

Number of simulations	100	250	500	1000	30s
DM vs BL	98.6% ±1.8%	99.1% ±1.1%	97.8% 1.6%	95.9% ±1.5%	
DM + ADM vs BL	80.1% ±1.2%	81.3% 2%	82.4 ±1.7%	85% ±1.4%	83.2% ±4%
DM + ADM vs DM	49.3% ±1.5%	56.1% ±1.9%	66.6% ±1.9%	78.1% ±1.1%	90.2% ±2%

The main limitation is that in some games, decisive moves have a marginal impact; for examples, in the game of Go, only in the very end of "Yose" (end games) such moves might occur (and resign usually happens much before such moves exist).

Further work

Extending decisive moves to moves which provide a sure win within M moves, or establishing that this is definitely too expensive, would be an interesting further work. We have just shown that for $M = 1$, this is not so expensive if we have a relevant data structure (we keep the $O(T \log(T))$).

A further work is the analysis of the complexity of anti-decisive moves, consisting in playing a move which forbids an immediate victory of the opponent. Maybe backtracking when a player wins and the loss was available would be a computationally faster alternative to antidecisive moves.

Decisive moves naturally lead to proved situations, in which the result of the game is known and fixed; it would be natural to modify the UCB formula in order to reduce the uncertainty term (to 0 possibly) when all children moves are proved, and to propagate the information up in the tree. To the best of our knowledge, there's no work around this in the published literature and this might extend UCT to cases in which perfect play can be reached by exact solving.

Acknowledgments

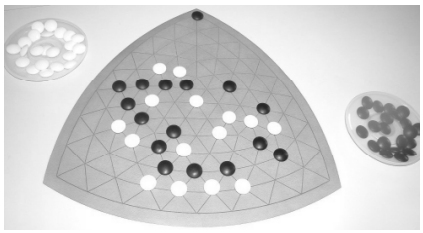
This work was supported by the French National Research Agency (ANR) through COSINUS program (project EXPLO-RA ANR-08-COSI-004).

References

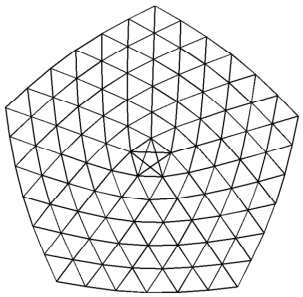
[1] J.-Y. Audibert, R. Munos, and C. Szepesvari. Use of variance estimation in the multi-armed bandit problem. In *NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation*, 2006.

[2] P. Auer. Using confidence bounds for exploitation-exploration tradeoffs. *The Journal of Machine Learning Research*, 3:397–422, 2003.

Hex



Y



*star

Fig. 3. Connection games (from wikipedia commons). In these games, two players play by filling in turn one location with one of their own stones. In Hex, the locations are hexagonal on a rhombus 11x11 board and each player has two sides and one color; the player who connects his two sides with his color has won. In Y, each player has a color and the player who connects three sides with his own color wins. In *Star, the game is finished when all locations are filled; then, all groups who do not contain at least two locations on the perimeter are removed, and the score of the game of a player is the sum of the scores of his groups of connected stones; and the score of a group is the number of locations of the perimeter that it contains, minus 4 (therefore, a group can have a negative score) - the player with the highest score has won and in case of draw the player who has more corners has won.

[3] A. Auger and O. Teytaud. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, Accepted.

[4] B. Bruegmann. Monte-Carlo Go. *Unpublished*, 1993.

[5] T. Cazenave. Nested monte-carlo search. In C. Boutilier, editor, *IJCAI*, pages 456–461, 2009.

[6] T. Cazenave and A. Saffidine. Utilisation de la recherche arborescente monte-carlo au hex. *Revue d'Intelligence Artificielle*, 23(2-3):183–202, 2009.

[7] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona Espagne, 2009. Springer.

[8] G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[9] G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.

[10] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, pages 72–83, 2006.

[11] R. Coulom. Computing "elo ratings" of move patterns in the game of go. *ICGA Journal*, 30(4):198–208, 2007.

[12] M. Crasmaru. On the complexity of Tsume-Go. 1558:222–231, 1999.

[13] M. Crasmaru and J. Tromp. Ladders are PSPACE-complete. In *Computers and Games*, pages 241–249, 2000.

[14] F. De Mesmay, A. Rimmel, Y. Voronenko, and M. Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. In *ICML*, Montréal Canada, 2009.

[15] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991.

[16] S. Gelly and D. Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.

[17] L. R. Harris. The heuristic search and the game of chess - a study of quiescene, sacrifices, and plan oriented play. In *IJCAI*, pages 334–339, 1975.

[18] L. Kocsis and C. Szepesvari. Bandit based monte-carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[19] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[20] C.-S. Lee, M.-H. Wang, G. Chaslot, J.-B. Hoock, A. Rimmel, O. Teytaud, S.-R. Tsai, S.-C. Hsu, and T.-P. Hong. The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, 2009.

[21] R. J. Lorentz. Amazons discover monte-carlo. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] J. Nash. Some games and machines for playing them. Technical Report D-1164, Rand Corporation, 1952.

[23] Reisch. Hex is PSPACE-complete. *ACTAINF: Acta Informatica*, 15:167–191, 1981.

[24] J. M. Robson. The complexity of go. In *IFIP Congress*, pages 413–417, 1983.

[25] P. Rolet, M. Sebag, and O. Teytaud. Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*, pages 302–317, 2009.

[26] M. P. D. Schadd, M. H. M. Winands, H. J. van den Herik, G. Chaslot, and J. W. H. M. Uiterwijk. Single-player monte-carlo tree search. In H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2008.

[27] R. W. Schmittberger. *New Rules for Classic Games*. Wiley, 1992.

[28] S. Sharma, Z. Kobti, and S. Goodwin. Knowledge generation for improving simulations in uct for general game playing. pages 49–55. 2008.

[29] F. Teytaud and O. Teytaud. Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, pages 65–74, Pamplona Espagne, 2009.

[30] Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21, 2008.

[31] Y. Wang and S. Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182, 2007.

[32] Wikipedia. Havannah, 2009.

[33] S. Zilberstein. Resource-bounded reasoning in intelligent systems. *Computing Surveys*, 28(4), 1996.