Combinatorial Games: From Theoretical Solving to AI Algorithms

Eric Duchêne(✉)

Université de Lyon, CNRS Université Lyon 1, LIRIS, UMR5205, 69622 Lyon, France
eric.duchene@univ-lyon1.fr

Abstract. Combinatorial game solving is a research field that is frequently highlighted each time a program defeats the best human player: Deep Blue (IBM) vs Kasparov for Chess in 1997, and Alpha Go (Google) vs Lee Sedol for the game of Go in 2016. But what is hidden behind these success stories? First of all, I will consider combinatorial games from a theoretical point of view. We will see how to proceed to properly define and deal with the concepts of outcome, value, and winning strategy. Are there some games for which an exact winning strategy can be expected? Unfortunately, the answer is no in many cases (including some of the most famous ones like Go, Othello, Chess or Checkers), as exact game solving belongs to the problems of highest complexity. Therefore, finding out an effective approximate strategy has highly motivated the community of AI researchers. In the current survey, the basics of the best AI programs will be presented, and in particular the well-known *Minimax* and *Monte-Carlo Tree Search* approaches.

1 Combinatorial Games

1.1 Introduction

Playing combinatorial games is a common activity for the general public. Indeed, the games of Go, Chess or Checkers are rather familiar to all of us. However, the underlying mathematical theory that enables to compute the winner of a given game, or more generally, to build a sequence of winning moves, is rather recent. It was settled by Berlekamp, Conway and Guy only in the late 70s [2,8]. The current section will present the highlights of this beautiful theory.

In order to avoid any confusion, first note that *combinatorial game theory* (here shortened as *CGT*) is very different from the so-called "economic" game theory introduced by Von Neumann and Morgenstern. I often consider that a preliminary activity to tackle CGT issues is the reading of Siegel's book [31] which gives a strong and formal background on CGT. Strictly speaking, a combinatorial game must satisfy the following criteria:

Definition 1 (Combinatorial Game). *In a combinatorial game, the following constraints are satisfied:*

Supported by the ANR-14-CE25-0006 project of the French National Research Agency and the CNRS PICS-07315 project.

© Springer International Publishing Switzerland 2016
S. Schockaert and P. Senellart (Eds.): SUM 2016, LNAI 9858, pp. 3–17, 2016.
DOI: 10.1007/978-3-319-45856-4_1

- *There are exactly two players, called "Left" and "Right", who alternate moves. Nobody can miss his turn.*
- *There is no hidden information: all possible moves are known to both players.*
- *There are no chance moves such as rolling dice or shuffling cards.*
- *The rules are defined in such a way that play will always come to an end.*
- *The last move determines the winner: in the* normal play convention*, the first player unable to move loses. In the* misère play convention*, the last player to move loses.*

Examples of such games are Nim [6] or Domineering [20]. In the first one, game positions are tuples of non-negative integers $(a_1, \dots, a_n)$. A move consists in strictly decreasing exactly one of the values a_i for some $1 \leq i \leq n$, provided the resulting position remains valid. The first player unable to move loses. In other words, reaching the position $(0, \dots, 0)$ is a winning move. The game Domineering is played on a rectangular grid. The two players alternately place a domino on the grid under the following condition: Left must place his dominoes vertically and Right horizontally. Once again, the first player unable to place a domino loses. Figure 1 illustrates a position for this game, where Left started and wins, since Right cannot place any additional horizontal domino.

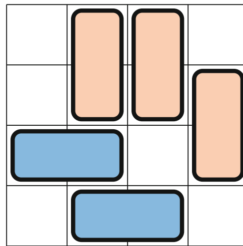

Fig. 1. Playing Domineering: right cannot play and loses

A useful property derived from Definition 1 is that any combinatorial game can be played indifferently on a particular (finite) tree. This tree is built as described in Definition 2.

Definition 2 (Game Tree). *Given a game $\mathcal{G}$ with starting position S, the game tree associated to $(\mathcal{G},\ S)$ is a semi-ordered rooted tree defined as follows:*

- *The vertex root correspond to the starting position S.*
- *All the game positions reachable for Left (resp. Right) in a single move from S are set as left (resp. right) children of the root.*
- *Apply the previous rule recursively for each child.*

Figure 2 gives an example of such a game tree for Domineering with starting position . For more convenience, note that only the top three levels of the tree are depicted (there is one additional level when fully expanded).

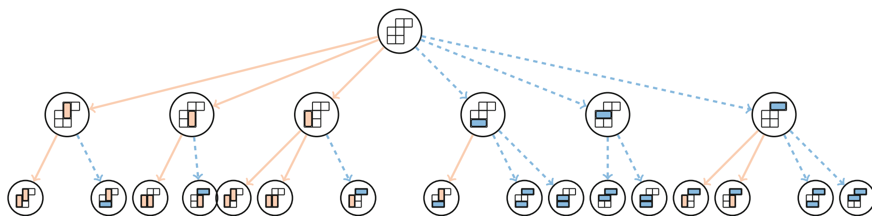

Fig. 2. Game tree of a DOMINEERING position

Now, playing any game on its game tree consists is moving alternately a token from the root to a leaf. Each player must follow an edge corresponding to his direction (i.e., full edges for Left and dashed ones for Right). In the normal play convention, the first player who moves the token on a leaf of the tree is the winner. We will see later on that this tree representation is very useful, both to compute exact and approximate strategies.

In view of Definition 1, one can remark that the specified conditions are too strong to cover some of the well-known abstract 2-player games. For example, Chess and Checkers may have draw outcomes, which is not allowed in a combinatorial game. This is due to the fact that some game positions can be visited several times during the play. Such games are called *loopy*. In games like Go, Dots and Boxes or Othello, the winner is determined with a score and not according to the player making the last move. However, such games remain very close to combinatorial games. Some keys can be found in the literature to deal with their resolution ([31], chap. 6 for loopy games, and [24] for an overview on scoring game theory). In addition, first attempts to built an "absolute" theory that would cover normal and misère play conventions, loopy and scoring games have been recently made [23]. Note that the concepts and issues that will be introduced in the current survey make also sense in this extended framework.

1.2 Main Issues in CGT

Given a game, researchers in CGT are generally concerned with the following three issues:

– Who is the winner?
– What is the value of a game (in the sense of Conway)?
– Can one provide a winning strategy, i.e., a sequence of optimal moves for the winner whatever his opponent's moves are?

For each of the above questions, I will give some parts of answer relative to the known theory.

The first problem is the determination of the winner of a given game, also called *outcome*. In a strict combinatorial game (i.e., a game satisfying the conditions of Definition 1), there are only four possible outcomes [31]:

- $\mathcal{L}$ if Left has a winning strategy independently of who starts the game,
- $\mathcal{R}$ if Right has a winning strategy independently of who starts the game,
- $\mathcal{N}$ if the first player has a winning strategy,
- $\mathcal{P}$ if the second player has a winning strategy.

This property can be easily deduced from the game tree, by labeling the vertices from the leaves to the root. Consequently, such an algorithm allows to compute the outcome of a game in polynomial time in the size of the tree. Yet, a game position has often a smaller input size than the size of its corresponding game tree. For example, a position $(a_1, \ldots, a_n)$ of Nim has an input size $\mathcal{O}(\sum_{i=1}^{n} \log_2(a_i))$, which is far smaller than the number of positions in the game tree. Hence, computing the whole game tree is generally not the good key to determine effectively the answer to Problem 1 below.

Problem 1 (Outcome). Given a game $\mathcal{G}$ with a starting position S, compute the complexity of deciding whether $(\mathcal{G}, S)$ is $\mathcal{P}$, $\mathcal{N}$, $\mathcal{L}$ or $\mathcal{R}$?

Note that for loopy games, the outcome *Draw* is added to the list of the possible outcomes.

Example 1. *The game* Domineering *played on a* 3×1 *grid is clearly* $\mathcal{L}$ *since there is no available (horizontal) move for Right. On a* 3×2 *and* 3×3 *grids, one can quickly check that the first player has a winning strategy. Such positions are thus* $\mathcal{N}$. *When* $n > 3$, *it can also be easily proved that* $3 \times n$ *grids are* $\mathcal{R}$, *since placing an horizontal domino in the middle row allows two free moves for Right, whereas a vertical move do not constraint further moves of Left.*

We now present a second major issue in CGT that can be considered as a refinement of the previous one.

Problem 2 (Value). Given a game $\mathcal{G}$ with a starting position S, compute its Conway's value.

The concept of game value was first defined by Conway in [8]. In his theory, each game position is assigned a numeric value among the set of *surreal numbers*. Roughly speaking, it corresponds to the number of moves ahead that Left has towards his opponent. For instance, position of Domineering has value -2 since Right can place two more dominoes than Left before being blocked. A more formal definition can be found in [31]. Just note that Conway's values are defined recursively and can also be computed from the game tree.

Knowing the value of a game allows to deduce its outcome. For example, all games having a strictly positive value are $\mathcal{L}$ and all games having a zero value are $\mathcal{P}$. Moreover, its knowledge is even more paramount when the game splits in sums: it means that a game $\mathcal{G}$ can be considered as a set of independent smaller games whose values allows to compute the overall value of $\mathcal{G}$. Consider the example depicted by Fig. 3. This game position can be considered as a sum of the three components , and of respective outcomes $\mathcal{L}$, $\mathcal{L}$ and $\mathcal{R}$, and respective Conway's values $1/2$, $1/2$ and -1. From this decomposition, there

is no way to compute the outcome of the general position from the outcomes of each component. Indeed, the sum of three components having outcomes $\mathcal{L}$, $\mathcal{L}$, and $\mathcal{R}$can either be $\mathcal{L}$, $\mathcal{R}$, $\mathcal{P}$ or $\mathcal{N}$. However, the sum of the three values can be easily computed and equals 0: we can conclude that the overall position of Fig. 3 is $\mathcal{P}$.

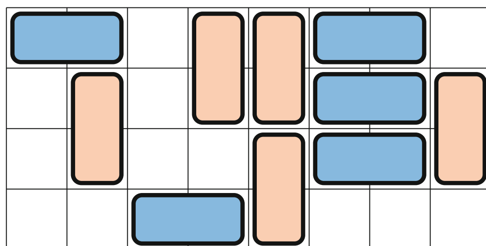

Fig. 3. Sum of DOMINEERING positions

Example 2. *Computing Conway's values of* DOMINEERING *is not easy even for small grids and there is no known formula to get them. On the other hand, the case of the game of* NIM *is better known. Indeed, Conway's value of any position* $(a_1, \ldots, a_n)$ *is an infinitesimal surreal number equal to* $a_1 \oplus \ldots \oplus a_n$*, where* $\oplus$ *is the bitwise XOR operator.*

The last problem is generally considered once (at least) the first one is solved.

Problem 3 (Winning Strategy). Given a game $\mathcal{G}$ and a starting position S, give a winning move from S for the player having a winning strategy. Do it recursively whatever the answer of the other player is.

There are really few games for which this question can be solved with a polynomial time algorithm. The game of NIM is one of them.

Example 3. *A winning strategy is known for the game of* NIM*: from any position* $(a_1, \ldots, a_n)$ *of outcome* $\mathcal{N}$*, there always exists a greedy algorithm that yields to a position* $(a'_1, \ldots, a'_n)$ *whose bitwise sum* $a'_1 \oplus \ldots \oplus a'_n$ *equals* 0 *(meaning that it will be losing for the other player).*

2 Complexity of Combinatorial Games

The complexity of combinatorial games is correlated to the computational complexity of the above problems. First of all, one can notice that all these problems are decidable, since it suffices to consider a simple algorithm on the game tree to have an answer. Of course, the size of the game tree remains an obstacle compared with the size of a game position. In [18], Fraenkel claims a game $\mathcal{G}$ is *polynomial* if:

- Problems 1 and 3 can be solved in polynomial time for any starting position S of $\mathcal{G}$.
- Winning strategies in $\mathcal{G}$ can be consumed in at most an exponential number of moves.
- These two properties remain valid for any sum of two game positions of $\mathcal{G}$.

If this definition is not always considered as a standard by the CGT community, there is a general agreement to say that the computational complexities of Problems 1 and 3 are the main criteria to evaluate the overall complexity of a game. Of course, this question makes sense only for games whose positions depends on some parameters such as the size of a grid, the values in a tuple... This explains why many famous games have been defined in the literature in a generalized version (e.g. Chess, Go, Checkers on a $n \times n$ board...). For almost all of them, even the computational complexity of Problem 1 is very high, as shown by Table 1 (extracted from [5,21]). Note that the belonging to class PSPACE or EXPTIME depends on the length of the play (exponential for EXPTIME and polynomial for PSPACE).

Table 1. Complexity of well-known games in their generalized versions

Game	Complexity
Tic Tac Toe	PSPACE-complete
Othello	PSPACE-complete
Hex	PSPACE-complete
Amazons	PSPACE-complete
Checkers	EXPTIME-complete
Chess	EXPTIME-complete
Go	EXPTIME-complete

In addition to these well-known games, there are many other combinatorial games that have been proved to be at least PSPACE-hard: Node-Kayles and Snort [28], many variations on Geography [25] or many other games on graphs. In 2009, Demaine and Hearn wrote a rich book about the complexity of many combinatorial games and puzzles [16]. If this list confirms that games belong to decision problems of highest complexity, some of them admit a lower one. The game of Nim is one of them and is luckily not the only one. For example, many games played on tuples of integers admit a polynomial winning strategy derived from tools arising from arithmetic, algebra or combinatorics on words. See the recent survey [11] which summarizes some of these games. Moreover, some games on graphs proved to be PSPACE-complete have a more affordable complexity on particular families of graphs. For example, Node Kayles is proved to be polynomial on paths and cographs [4]. This is also the case for Geography played on undirected graphs [19]. Finally, note that the complexity of Domineering is still an open problem.

If the computational complexity of many games is often very high, it makes no sense to consider it when the game positions have a constant size. It is in particular the case for well-known board games such as Chess on a 8×8 board, the game of Go on a 19×19 board, or standard Hex. Solving them is often a question a computational performance and algorithmic optimization on the game tree. In this context, these games can be classified according to the status of their resolution. For that purpose, Allis [1] defined three levels of resolution for a game:

- *ultra-weakly solved*: the answer of Problem 1 is known, but Problem 3 remains open. This is for instance the case of Hex, that is winning for the first player, but no winning strategy has been computed yet.
- *weakly solved*: Problems 1 and 3 are solved for the standard starting position (e.g., standard initial position of Checkers, empty board of Tic Tac Toe). As a consequence, the known winning strategy is not improved if the opponent does not play optimally.
- *strongly solved*: Problems 1 and 3 are solved for any starting position.

According to this definition, Table 2 summarizes the current knowledge about the resolution of some games.

Table 2. Status of the resolutions of several well-known games

Game	Size of the board	Resolution status
Tic Tac Toe	3×3	Strong
Connect Four	6×7	Strong
Checkers	8×8	Weak
Hex	11×11	Ultra-Weak
Go	19×19	Open
Chess	8×8	Open
Othello	8×8	Open

A natural question arises when reading the above table. What makes a game harder than another one? If there is obviously no universal answer, Fraenkel suggests several relevant criteria in [17].

- The average *branching factor*, i.e., the average number of available moves from a position (around 35 for Chess and 250 for the game of Go).
- The total number of game positions (10^{18} for Checkers, 10^{171} for the game of Go).
- The existence of cycles. In other words, loopy games are harder than non loopy ones.

- *Impartial* or *Partizan*. A game is said *impartial* if both players always have the same available moves. It implies the game tree to be symmetric. Nim is an example of an impartial game, whereas Domineering and all the games mentioned in Table 2 are not. Such games are called partizan. Impartial games are in general easier to solve since their Conway's values are more "controlled".
- The fact that the game can be decomposed into sums of smaller independent games (as it is the case for Domineering).
- The number of final positions.

Based on these considerations, how to deal with games whose complexity is too high - either theoretically, or simply in view of their empirical hardness? Approximate resolutions (especially for Problem 3) must be considered and artificial intelligence algorithms were introduced to this end.

3 AI Algorithms to Deal with the Hardest Games

In the previous section, we have seen that Problem 1 remains unsolved for games having a huge number of positions. If the recent work of Schaeffer et al. [29] on Checkers was a real breakthrough (they found the exact outcome, which is a Draw), getting a similar result for games like Chess, Othello or Go seems currently out of reach. Moreover, researchers generally feel more concerned by finding a good way to play these games rather than computing the exact outcome. In the 50s, this interest led to the beginnings of artificial intelligence [30] and the construction of the first programs to play Chess [3]. For more information about computer game history, see [27]. Before going into more details on AI programs for games, note that in general, these algorithms work on a slight variation of the game tree given in Definition 2, where Left is always supposed to be the first player, and only the moves of one player are represented on a level of the tree. For example, the children of the root correspond exclusively to the moves available for Left, their children to the possible answers for Right...

3.1 MiniMax Algorithms

The first steps in combinatorial game programming were made for Chess. The so-called *MiniMax* approach is due to Shannon and Turing in the 50 s and has been widely considered in many other AI programs. Its main objective is to minimize the maximum loss of each player. This algorithm requires some expert knowledge of the game, as it uses an evaluation function of the values of game positions.

Roughly speaking, in a MiniMax algorithm, the game tree is built up to a certain depth. Then each leaf of this partial game tree is evaluated thanks to an evaluation function. This function is the key of the algorithm and is based on heuristic considerations. For example, the Chess computer *Deep Blue* (who first defeated a human world champion in 1996) had an evaluation function based on hundreds of parameters (e.g. compare the power of a non-blocked tower versus

a protected king). These parameters were tuned after an fine analyze of 700,000 master games. Each parent node of a leaf is then assigned a value equals to the minimum value of its children (wlog, we here assume that the depth is even - then the last moves correspond to moves for Right, whose goal is to minimize the game value). The next parent nodes are evaluated by taking the maximum value among their children (it corresponds to moves for Left). Then recursively each parent node is evaluate according to the values of its children, by taking alternately the minimum or the maximum according to whether it is Left or Right's turn. Figure 4 illustrates this algorithm on a tree of depth 4. In this example, assume an evaluation function provides the values located on the leaves of the tree. Then MiniMax ensures that Left can force a win with a score equals to 4. Red nodes are those for which the maximum of the children is taken, i.e., positions from which Left has to play.

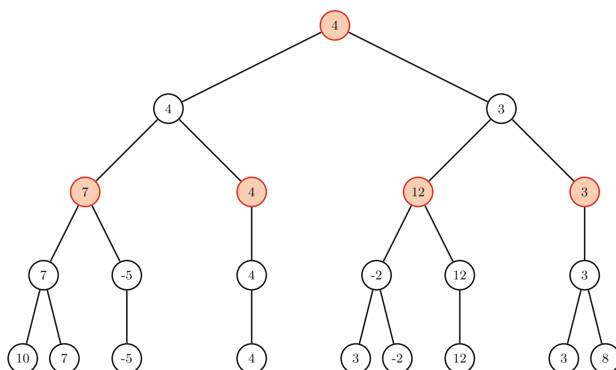

Fig. 4. MinMax algorithm on a tree of depth 4

In addition to an expert tuning of the evaluation function, another significant enhancement was made with the introduction of *Alpha-Beta pruning* [12]. It consists in a very effective selective cut-off of the Minimax algorithm without loss of information. Indeed, if after having computed the values of the first branches, it turns out that the overall value of the root is at least v, then one can prune all the unexplored branches whose values are guaranteed to be less than v. The ordering of the branches in the game tree then turns out to be paramount, as it can considerably increase the efficiency of the algorithm. In addition to this technique, one can also mention the use of *transposition tables* (adjoined to alpha-beta pruning) to speed up the search in the game tree.

Nowadays, the MiniMax algorithm (together with its improving techniques) is still used by the best algorithms to solve games admitting a relevant evaluation function. This is for example the case for Chess, Checkers, Connect Four or Othello. Yet, we will see that for other games, some probabilistic approaches turn out to be more efficient.

3.2 Monte-Carlo Approaches

In 2006, Coulom [9] suggested to combine the principle of the MiniMax algorithm with Monte Carlo methods. These methods were formalized in the 40 s to deal with hard problems by taking a random sampling. For example, they can be used to estimate the value of π. Of course, the quality of the approximated solution partially depends on the size of the sample. In our case, their application will consist in simulating many random games.

The combination of both MiniMax and Monte Carlo methods is called MCTS, which stands for *Monte Carlo Tree Search*. Since its introduction, it has been considered by much research on AI for games. This success is mainly explained by the significant improvements made by computer Go programs that are using this technique. Moreover, it has also shown very good performances for problems for which other techniques had poor ones (e.g. some problems in combinatorial optimization, puzzles, multi-player games, scheduling, operation research...). Another great advantage of MCTS is that there is no need of a strong expert knowledge to implement a good algorithm. Hence it can be considered for problems for which humans do not have a strong background. In addition, MCTS can be stopped at any time to provide the current best solution and the tree built so far can be reused for the next step.

In what follows, we will give the necessary information to understand the essence of MCTS applied to games. For additional material, the reader could refer to the more exhaustive survey [7].

The basic MCTS algorithm consists in building progressively the game tree, guided by the results of the previous explorations of it. Unlike the standard MiniMax algorithm, the tree is built in an asymmetric manner. The in-depth search is considered only for the most promising branches that are chosen according to a tuned selection policy. This policy relies on the values of each node of the tree. Roughly speaking, the value of a node v_i corresponds to the percentage of winning random simulations when v_i is played. Of course this value become more and more accurate when the tree grows.

Description. As illustrated in Fig. 5, each iteration of MCTS is organized around 4 steps called *descent, growth, roll-out* and *update*. Numbers in grey correspond to the estimate values of each node (a function of the pourcentage of win). Here are their main description:

- Descent: starting from the root of the game tree, a child is recursively selected according to the selection policy. As seen on the figure, a MiniMax selection is used to descend the tree, according to the values of each node (here, $B1$ is the most promising move for Left, then $E1$ for Right). This descent stops when it lands on a node that needs to be expanded (also given by the policy). In our example, the node $E1$ is such a node.
- Growth: Add one or more children to this expandable node in the tree. On Fig. 5, Node $B4$ is added to the tree.

– Rollout: From an added node, make a simulation by playing random moves until the end of the game. In our example, the random simulation from $B4$ leads to a loss for Left.
– Update: the result of the simulation is backpropagated to the moves of the tree that have been selected. Their values are thus updated.

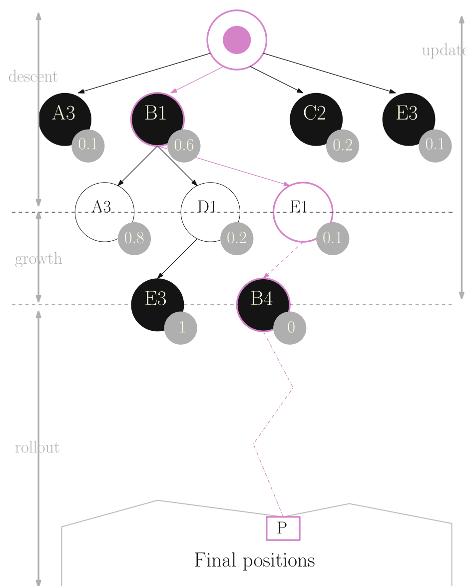

Fig. 5. The four stages of the MCTS algorithm

Improvements. In general, MCTS is not used in a raw version and is frequently combined with additional features. As detailed in [36], there is a very rich literature on the improvements brought to MCTS. They can be organized according to the stage they impact. Table 3 summarizes the most important enhancements brought to MCTS.

One of the most important feature of the algorithm is the node selection policy during the descent. At each step of this stage, MCTS chooses the node that maximizes (or minimizes, according to whether it is Left or Right's turn) some quantity. A formula that is frequently used is called *Upper Confidence Bounds* (UCB). It associates to each node v_i of the tree the following value:

$$V(v_i) + C \times \sqrt{\frac{\ln N}{n_i}},$$

where $V(v_i)$ is the percentage of winning simulations involving v_i, n_i is the total number of simulations involving v_i, N is the number of times its parent has been visited, and C is a tunable parameter. This formula is well-known in the context of bandit problems (choose sequentially amongst n actions the best one

Table 3. Main improvements brought to MCTS

Stage	Improvement
Descent	UCT (2006) [22]
Descent	RAVE (2007) [15]
Descent	Criticality (2009) [10]
Growth	FPU (2007) [35]
Rollout	Pool-RAVE (2011), [26]
Rollout	NST (2012) [33]
Rollout	BHRF (2016) [14]
Update	Fuego reward (2010) [13]

in order to maximize the cumulative reward). It allows in particular to deal with the exploration-exploitation dilemma, i.e., to find a balance between exploring unvisited nodes and reinforce the statistics of the best ones. The combination of MCTS and UCB is called UCT [22].

A second common enhancement for MCTS during the descent is the RAVE estimator (Rapide Action-Value Estimator [15]). It consists in considering each move of the rollout as important as the first move. In other words, the moves visited during the rollout stage will also affect the values of the same moves in the tree. On Fig. 5, imagine the move $E3$ is played during the simulation depicted with dashed line. Then RAVE will thus modify the UCB value of the node $E3$ of the tree (the RAVE formula will not be given here).

MCTS has also been widely studied in order to increase the quality of the random simulations. A first way to mimic the strategy of a good player is to consider evaluations functions based on expert knowledge. In [34], moves are categorized according to several criteria: location on the board, capturing or blocking potential and proximity to the last move. Then the approach is to evaluate the probability that a move belonging to a category will be played by a real player. This probability is determined by analyzing a huge sample of real games played by either humans or computers. Of course this strategy is fully specific to the game on which MCTS is applied. More generic approaches were considered such as *NST* [33], *BHRF* [14] or *Pool RAVE* [26]. In the first two ones, good sequences of moves are kept in memory. Indeed, it is rather frequent that given successive attacking moves of a player, there is an usual sequence of answers of the opponent to defend himself. In the second one, the random rollout policy is biased by the values in the game tree, i.e., good moves visited in the tree are likely to be played during a simulation.

In addition to the enhancements applied to the different stages of MCTS, one can also mention several studies to parallelize the algorithm that perform very good results [36].

We cannot conclude this survey without mentioning the outstanding performances of Google's program *Alpha Go* [32]. Like *Deep Blue* for Chess, Alpha

Go is the first AI to defeat the best human player in Go. This program runs an MCTS algorithm combined with two deep neural networks. The first one is called the *Policy network* and is used during the descent phase to find out the most promising moves. It was bootstrapped from many games of human experts (around 30 million moves analyzed during three weeks on 50 GPU). The reinforcement learning was then enhanced by many games of self-play. The second neural network is called *Value network* and can be considered as the first powerful evaluation function for Go that is used to bias the rollout policy. If Alpha Go's performances show a real breakthrough in AI programs for games, the last day of this research field has not yet come. In particular, the need of expert knowledge to bootstrap the networks cannot be considered when dealing with problems for which humans have a poor expertise.

4 Perspectives

Working on problems as hard as combinatorial games is a real challenge, both for CGT and AI researchers. The major results obtained in the past years are very stimulating and encourage many people to strengthen the overall effort on the topic. Hence, from a theoretical point of view, the next step for CGT is the construction of a general framework to cope with scoring games. In particular, the question of the sum of two scoring games is paramount, as it is radically different from the sum games in normal play convention (one cannot simply add the values of each game). First attempts have been recently made in that sense to consider Conway's values as waiting moves in scoring games.

Concerning AI algorithms for games, as said in the above paragraph, Alpha Go has been a breakthrough for the area but very exciting issues remain. More precisely, the neural network approach proposed by Google requires a wide set of expert knowledge and needs computer power for a long time. However, there are some games for which both are not available. In particular, the example of *General Game Playing* is a real challenge for AI algorithms, as the rules of the game are given at the latest 20 minutes before running the program. Supervised learning techniques like those of Alpha Go are thus almost impossible to set up, and standard MCTS enhancements are currently the most effective ones for this kind of problem. In addition, one can also look for adapting MCTS to problems of higher uncertainty such as multi-player games or games having randomness in their rules (use of dices for example). First results have already been made in that direction [36].

References

1. Allis, L.V.: Searching for solutions in games an artificial intelligence. Ph.D. Maastricht, Limburg University, Netherland (1994)
2. Berlekamp, E., Conway, J.H., Guy, R.K.: Winning ways for your mathematical plays, vol. 1, 2nd edn. A K Peters Ltd., Natick (2001)
3. Bernstein, A., Roberts, M.: Computer V. Chess player. Sci. Am. **198**, 96–105 (1958)

4. Bodlaender, H.L., Kratsch, D.: Kayles and numbers. J. Algorithms **43**, 106–119 (2002)
5. Bonnet, E., Saffidine, A.: Complexit des Jeux (in french). Bulletin de la ROADEF **31**, 9–12 (2014)
6. Bouton, C.L.: Nim, a game with a complete mathematical theory. Ann. Math. **3**, 35–39 (1905)
7. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Trans. Comput. Intell. AI Games **4**(1), 1–43 (2012)
8. Conway, J.H.: On Number and Games. Academic Press Inc., Cambridge (1976)
9. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: Proceedings of the 5th International Conference on Computers and Games, Turin, Italy, pp. 72–83 (2006)
10. Coulom, R.: Criticality: a Monte-Carlo Heuristic for Go Programs. Invited talk in University of Electro-Communication, Tokyo (2009)
11. Duchêne, E., Fraenkel, A.S., Gurvich, V., Ho, N.B., Kimberling, C., Larsson, U.: Wythoff Wisdom (preprint)
12. Edwards, D.J., Hart, T.P.: The $\alpha - \beta$ heuristic. In: Artificial intelligence project RLE and MIT Computation Centre, Memo 30 (1963)
13. Enzenberger, M., Muller, M., Arneson, B., Segal, R.: Fuego an open-source framework for board games and Go engine based on Monte Carlo tree search. IEEE Trans. Comput. Intell. AI Games **2**(4), 259–270 (2010)
14. Fabbri, A., Armetta, F., Duchne, E., Hassas, S.: A self-acquiring knowledge process for MCTS. Int. J. Artif. Intell. Tools **25**(1), 1660007 (2016)
15. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: Ghahramani, Z. (ed.) Proceedings of the International Conference on Machine Learning (ICML), pp. 273–280. ACM, New York (2007)
16. Hearn, R.A., Demaine, E.D.: Games, Puzzles, and Computation. A K Peters (2009)
17. Fraenkel, A.S.: Nim is easy, chess is hard - but why? J. Int. Comput. Games Assoc. **29**, 203–206 (2006)
18. Fraenkel, A.S.: Complexity, appeal and challenges of combinatorial games. Theor. Comput. Sci. **313**, 393–415 (2004)
19. Fraenkel, A.S., Simonson, S.: Geography. Theor. Comput. Sci. **110**, 197–214 (1993)
20. Gardner, M.: Mathematical games: cram, crosscram and quadraphage: new games having elusive winning strategies. Sci. Am. **230**, 106–108 (1974)
21. Junghanns, A., Schaeffer, J.: Sokoban: enhancing general single-agent search methods using domain knowledge. Artif. Intell. **129**(1), 219–251 (2001)
22. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS (LNAI), vol. 4212, pp. 282–293. Springer, Heidelberg (2006)
23. Larsson, U., Nowakowski, R.J., Santos, C.: Theory, Absolute Combinatorial Game. arXiv:1606.01975 (2016)
24. Larsson, U., Nowakowski, R.J., Santos, C.: When waiting moves you in scoring combinatorial games. arXiv:1505.01907 (2015)
25. Renault, G., Schmidt, S.: On the complexity of the misre version of three games played on graphs (preprint)
26. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo simulations through RAVE values. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2010. LNCS, vol. 6515, pp. 59–68. Springer, Heidelberg (2011)

27. Rougetet, L.: Combinatorial games and machines. In: Pisano, R. (ed.) A Bridge between Conceptual Frameworks, Sciences, Society and Technology Studies, Pisano, vol. 27, pp. 475–494. Springer, Dordrecht (2015)
28. Schaeffer, T.J.: On the complexity of some two-person perfect-information games. J. Comput. Syst. Sci. **16**, 185–225 (1978)
29. Schaeffer, J., Burch, N., Bjrnsson, Y., Kishimoto, A., Mller, M., Lake, R., Lu, P., Sutphen, S.: Checkers is solved. Science **317**(5844), 1518–1522 (2007)
30. Shannon, C.: Programming a computer for playing chess. Philos. Mag. Ser. 7 **41**(314), 256–275 (1950)
31. Siegel, A.N.: Combinatorial Game Theory. American Mathematical Society, San Francisco (2013)
32. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
33. Tak, M.J.W., Winands, M.H.M., Bjornsson, Y.: N-grams and the last-good-reply policy applied in general game playing. IEEE Trans. Comput. Intell. AI Games **4**(2), 73–83 (2012)
34. Tsuruoka, Y., Yokoyama, D., Chikayama, T.: Game-tree search algorithm based on realization probability. ICGA J. **25**(3), 132–144 (2002)
35. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawai, pp. 175–182 (2007)
36. Winands, M.: Monte-Carlo tree search in board games. In: Nakatsu, R., Rauterberg, M., Ciancarini, P. (eds.) Handbook of Digital Games and Entertainment Technologies, pp. 1–30. Springer, Heidelberg (2015)