An Account of a Participation to the 2007 General Game Playing Competition

Jean Mehat and **Tristan Cazenave**
LIASD, Dept. Informatique, Université Paris 8
2 rue de la liberté, 93526, Saint-Denis, France
jm@ai.univ-paris8.fr
cazenave@ai.univ-paris8.fr

Abstract

We describe the participation of our program to the 2007 General Game Playing Tournament. It finished at the third place of the qualifying tournament. After an informal description of General Game Playing and of its Game Description Language, we present in details the structure of our player as it participated to the qualifying phase of the tournament. We then present the context of this phase of the tournament and analyze the performance of our program. Finally, we present the way it was modified to participate in the final phase and give a qualitative assessment of its playing strength.

Introduction

General Game Playing tries to address the shortcomings of current specialized game playing programs that cannot adapt to other domains than the game they were programmed for. The goal is to find general algorithms for games, and to have more general intelligence than game-specific programs.

This paper describes the participation of the general game program Ary to the 2007 General Game Playing Tournament. The emphasis is on the participation to the competition. We also advocate the use of Monte-Carlo methods. The second section is about the Game Description Language. The third section details the structure of Ary. The fourth section compares Ary to the other competitors. The fifth section describes the tournament. The sixth section analyzes the performances of Ary. The seventh section deals with UCT.

The Game Description Language

The Game Description Language (GDL) is used to describe a game. It is based on first order logic, hence missing arithmetic. We describe it informally with a very simple example. We are using the KIF notation of the GDL language, that is reminiscent of the Lisp syntax.

The figure 1 contains a representation in GDL of a binary version of the simultaneous play game *My father has more money than yours* (Berlekamp, Conway, & Guy 1982)

The rules indicate that there are two players (*left* and *right*, enumerates the legal moves (*telling a figure*), identify the terminal nodes (*after the first and only move*) and the reward for each player: 0 for the smaller figure, 100 for the greatest and 50 in case of tie. The description of this game does not need the INIT keyword, used to describe the initial state of the board.

Table 1: The GDL representation of simultaneous play, binary version of the game *My father has more money that yours*.

```
(ROLE left) (ROLE right)       ; players
(LEGAL (DOES ?player (tell 0))); moves
(LEGAL (DOES ?player (tell 1)))
(<= (NEXT (value ?p ?x)) (DOES ?p (tell ?x)))
(<= TERMINAL (TRUE (value ?p ?x)))
(<= (other ?x ?y) (role ?x) (role ?y)
    (DISTINCT ?x ?y))
(<= (GOAL ?p 0) (TRUE (value ?p 0))
    (other ?p ?op) (TRUE (value ?op 1)))
(<= (GOAL ?p 50) (TRUE (value ?p ?x))
    (other ?p ?op) (TRUE (value ?op ?x)))
(<= (GOAL ?p 100) (TRUE (value ?p 1))
    (other ?p ?op) (TRUE (value ?op 0)))
```

In the following, we designate the current situation of the game as the *board status*, even for games that are not played on a board like the one described in 1.

The structure of Ary

The version of Ary that participated in the qualifying phase of the tournament is written in C. It consists in about 4300 lines of code, including comments and self-test support. It uses a Prolog interpreter as an inference engine.

Data structures used in Ary

The final objects are atoms, used to represent all the constituents of the game description: keywords of GDL, operators of the logic, predicates, names, integers and variables used in the description of the game. On its creation, each atom receives a random 64 bits hash key.

The final objects are combined in lists, represented with *Cons* cell containing two pointers on the head and tail of the list. Each Cons cell has also a hash code, obtained with a combination of the hash codes of its head and tail, that is used to compare quickly lists that differ (identical lists have to be compared in the EQ sense of Lisp, as different lists may have the same hash code) and find them in hash tables. We did not implement garbage collection: the core of the program identify lists that won't be referenced and add them explicitly to the free list.

Most of the lists are stored in tables of list. The table contains an array and a hash table. In the array, the elements are stored sequentially, giving an easy way to iterate over all the elements of the table. The hash table allows to quickly verify if an element is already present in the table. When a table is full, it is reallocated with a greater number of slots.

A *node* data structure was added for representing the move tree built by the UCT algorithm. It contains a table describing the current board status, a table of legal moves for each player, the current evaluation for each player, a marker to identify terminal nodes and when useful an array of pointers to the children nodes.

Interface with Prolog

After trying without much success to implement in C a quick pattern matcher that would be able to apply the rules of the games, we made the choice to use a Prolog Interpreter as an inference engine. We used SWI-Prolog because of its availability and good interface with the C language (Wielemaker 2003). We received a gentle help of the principal developer of SWI-Prolog when we encountered problems with overflow in the Prolog stack.

When translating the game description into its internal form, Ary makes a few transformations to ensure an easy interface with Prolog: characters that are invalid in Prolog names are replaced by something that the interpreter accepts, provision is taken to avoid name clashes with predicates already defined in the interpreter. Clauses in the right part of the theorems are re-ordinated to avoid problems encountered on various games descriptions and to try to obtain better performance: clauses without variables are made to come first, and the DISTINCT clauses are pushed back after the first clauses using its variables.

In certain game descriptions, there are predicates that are not recognized as such by the Prolog interpreter; for example, in our simple game, the predicate `value`, that is used only after the first (and only) move. To avoid an error of Prolog in these cases, each functor (in the Prolog sense) used in the game description is used to assert and then retract an arbitrary assertion with the correct arity.

After loading the game, it is translated into the Prolog language by a simple recursive descent that treats only specifically the implication and distinct forms of the GDL description. All of the other forms of the description as `(FOO A B C)` in internal form are simply rewritten in Prolog as `foo(a, b, c)`. Similarly, we have a function that is used to translate back the answers of the Prolog interpreter to the internal form.

The game description in Prolog is sent once for all to the interpreter, with the initial board status. The Prolog interpreter is then used to identify terminated games, find the scores of the players in these states, enumerate the legal moves and the consequences of moves on the board status, and to modify the image of the board status in the interpreter.

Terminated games are identified by making the Prolog interpreter try to demonstrate `terminal`. Scores are calculated by reading the answers to the `goal(X, Score)` question for each player and consulting the value of the `Score` variable.

Similarly, legal moves are enumerated by reading the answers of the Prolog interpreter to `does(X, Move)` for each player x and the new board status is found by asserting the moves, reading the values of the variable X in the answers to the `next(X)` question and retracting the moves. For these two questions, given the GDL description, the Prolog interpreter may return an answer many times; the identification and elimination of these duplicate cases is done in C.

The transition from one board status to another is done incrementally: Ary transmits to Prolog the retraction of what is present only in the old state, and asserts only what appears only in the new state. Unmodified characteristics of the board status do not incur exchange with the Prolog interpreter. So in board games like Tic Tac Toe where a move modifies the status of one cell described by one assertion only, the transition from one board status to the next is done with one retraction and one assertion.

Monte-Carlo implementation

The Monte-Carlo implementation is straightforward: until the expiration of the thinking time, the current board status is loaded into the interpreter, legal moves are generated, and a random game is played until arriving in a terminal board status. The score is asked to the interpreter and accumulated in a counter associated with each of the first legal moves.

When the thinking time expires, the current game is stopped and the move with the best mean for Ary is chosen. Ignoring the scores of the other players has a clear advantage in simplicity: it is not necessary to distinguish between games that have one, two or more players; zero-sum games and cooperative games are treated identically. Moreover, we expected it to provide more interesting play, and it was in agreement with our goal to have Ary plays its best moves. Finally, the round-robin nature of the qualifying phase made it uninteresting to try to limit the score of the opponent.

Due to lack of time, we did not make Ary use the initial thinking time to start the Monte-Carlo exploration to make a better choice on the first move.

Comparison with other competitors

On the eight other competitors, two have been described in published articles we are aware of.

Clune Player, developped by James Clune, won the 2005 tournament. It uses a classical alpha-beta search with transposition table. Its main innovation is to identify important features of the board state, like mobility of pieces, that are used to direct the search (Clune 2007)

Flux Player is developped primarily by Stephan Schiffe Michael Thielscher (Schiffel & Thielscher 2007). It is also based on alpha-beta search, and is primarily implemented in Prolog. We think that its most interesting in that, from the description of the rules, it computes a *distance* between the current board status and those of terminal states where the reward can be computed; this distance is used to obtain intermediate evaluations. FluxPlayer won the 2006 tournament.

From a private communication at the AAAI 2007, we know that CadiaPlayer, developped by Hilmar Finnsson and Yngvi Björnsson is based on UCT, like the version of Ary we prepared for the final phase of the competition, but little is known on the details of its architecture. CadiaPlayer is the winner of the 2007 competition.

The GGP AAAI 07 tournament

We describe in this section lessons learned from our participation to the 2007 tournament.

Communication with the organization

Before the tournament, the organizers gave access to a server where it was possible to load rules for various games and replay matches, including ones played in the preceding tournaments. It was also possible to set up games refereed by the official Game Master, but the lack of communication between competitors made it only possible to have Ary play single-player games or games against itself. The server includes a forum, but is not very active: at the beginning of January 2008, its last post was a question we asked in May 2007 that is still awaiting its answer.

Communication with the organizers was done primarily via a mailing list, used for general announcement. Every other playing day, the organizers posted a summary of the games played with the pairings and the score of the players.

Questions regarding particular situations went through personal mails. In the few situations of erroneous game rules, the problem were quickly identified, the games aborted, the rules corrected and the games restarted.

During the match, a special site was open, where it was possible to follow the games as they were played, represented with an intuitive graphic representation. It would have been a good idea to keep a copy of these clear representation of the matches with a pointer on the log files generated by Ary to facilitate post-tournament analysis.

Playing schedule and pairing

The qualifying tournament was played on four weeks of June, with two playing days per week.

The playing day started around 8pm CET, and lasted for 6 to ten hours, with each program playing about ten matches per day.

Each day included various games with varying initial and playing times, between 10 seconds and 10 minutes.

In two players games, each program usually played two games, both with the same opponent: one as first player and the other one as second player. This made it possible, once a game identified from its obfuscated form, to identify the opponent from the weekly summary that included the pairings for each game. When necessary for three and four players games, the organizers used random players for parity.

Obfuscation of the rules

The rules of the games were transmitted in an obfuscated form: the atoms, except the keywords, were replaced by arbitrary strings. For example instead of the theorem for incrementing a variable, usually noted as:

```
(<= (NEXT (step ?x))
      (TRUE (step ?y))
      (succ ?y ?x))
```

the players may receive:

```
(<= (NEXT (THISHAND ?YOUBTFAINKS))
      (TRUE (THISHAND ?THASTABOLAN))
      (POSEEP ?THASTABOLAN ?YOUBTFAINKS))
```

The obfuscation of the game was only partial: the integers, when used, were not obfuscated and the replacement of strings was done globally on every occurrence in the rules.

The non-obfuscation of integers means that the predicates referring to board cells could be identified at first sight from their description in the initial state in the many games where board cells are noted with integers. Would the integer have been obfuscated, the integers could still have been recognized using the successor relation with its well established pattern, but it would have been an heavier task. For games like Chinese checkers, were the board cells were named in the rules as `a1` or `f5`, it was more difficult to recognize cells at first sight as they were obfuscated.

The global replacement of strings means that the names of the variable could be used to facilitate the interpretation of predicates. In the original rules, variables are usually named after the type of data they contain. Once it is recognized, from a particular predicate that the variables `?TERFERVIR` and `?ITHICHANG` refer to the coordinates of a cell, they can be replaced by `?line` and `?row` in all of the rules, facilitating the interpretation of the other rules.

Ary did not make any use of these features of the obfuscation process, but they were very helpful when we had to analyze its comportment in post-game analysis, as it is usually easier for us to examine non-obfuscated games.

Program crashes and network lags

In case of failure of a player to give legal moves in the specified time, the Game Master played a random move for it. In the qualifying phase, the game was nonetheless scored as usual, while in the final phase, a player scored no point if the player missed more that one move.

The rationale behind this choice is that the opponent of a crashing program has to furnish some work to score points, but its side effect is that two programs crashing before the first move of Tic Tac Toe match would have scored identically as two programs playing perfectly. The decision for the final phase to let a program score only if it played until the end of the match seems more reasonable.

It is to be noted that the random moves played by the Game Master in place of a deficient program seem to actually have been the first one, in the order they were generated by our Prolog interpreter. Ary did not exploit this, but a program playing only to win could use this predictability to maximize its score.

Being situated in Europe, we had provided for net lags, in keeping a margin of one second before the falling of the clock to send the chosen move. From the presentation of the matches on the server, we think that the Game Master had itself a tolerance for networks lags by its side, but no detail was published on this point.

Communications between the player and the game master

The players communicate only with the Game Master, and the communication is reduced to the minimum: the first message contains only the rules of the game, the role of the player and the two values for the initial and playing time. Then each message contains only the moves of the players.

The post analysis of the matches would be easier if the Game Master transmitted, after the end of the match, the identity of the opponents. The name of the game in clear, the score for all the players and the moves where the Game Master supplied to deficiencies of the players would also be helpful.

Performance during the qualifying phase

Due to the terse nature of the Game Master, the analysis of the matches played by Ary presented here is entirely based on its log files, and may be incomplete or inaccurate. For example, the score obtained was not logged during the first day and we had to replay the moves played to deduce the score obtained.

During the whole qualifying phase, Ary ran on a PC with a 32 bits dual core 3GHz processor and 1 Gb of central memory. As allowed by the rules, the program was constantly modified during the competition: deficiencies were corrected as soon as possible.

On the following, we express the results of the games as in the competition with a number between 0 (lost) and 100 (win).

First week

The first day, the program played simple games. Ary failed only on the 3×3 puzzle, that it was not able to solve and got only 90 on beatmania. On two games players, it got 50 (a tie) on its four games of Tic Tac Toe, playing on one or two grids; the 2 players chinese checkers also lead to a tie, while it won the four players chinese checkers game. It lost its two games of blocker, as first and second player.

On the second day, Ary succeeded in not crashing its space ship, scoring 50 for the single player game of the day; as could be expected, the variation of the initial time between 10 and 60 seconds made no difference, as this time was not used.

The other games of the day were of the *cat and mouse* type, played on 8×8 grids with obstacles; in one of those ary won as the cat (maybe with the help of a crash of the other player at move 20) but lost as the mouse; in the other it won as the mouse and made a tie as the cat. In the simplified version of pacman, it won when playing as one of the ghost and escaped them in the other match, getting 74.

At the end of the first week, Ary was the third out of nine participants. That was quite a good surpise, as our participation was primarily intented to gain experience in future tournaments.

Second week

The third day of competition, the games were variations of the games of the first week, doubled in a parallel of serial way. For example, in parallel Tic Tac Toe, there are two boards and the program play one move on each of the grids, squaring the number of possible moves; in serial Tic Tac Toe, a match is played on one grid and when finished, another match is played on another grid. The final reward is the mean of the results of the two matches. Ary did pretty well, except on asteroids and on one of the variations of the Block World where it crashed while reading the rules because of an obscure bug in the parser.

On the fourth day, Ary did not perform as well as in the preceding days. The programs played stress tests presented as single player games. In one group, the player can always give up or continue, but the amount of efforts to prove it is legal to continue grows with game length (linearly, quadratically or exponentially). In the second group, the move tree has a large number of nodes (10^3, 10^6 or 10^9) that are mostly duplicates. In the last group, the programs had to search into a large tree (10^3, 10^6 or 10^9 nodes).

Ary crashed at the beginning of three of these games for stupid reasons: one of the set of rules was larger (about 85 kilo-bytes) that the maximum we had anticipated, and some predicates had more than 26 arguments. On the other games, it did not perform well. We think that the absence of transposition tables was costly.

After this second week, Ary was at the 5th place. We easily solved the error limiting the number of arguments of a predicate and we replaced nearly all statically allocated arrays by dynamical ones, reallocated when necessary.

Third week

On the fifth day, the programs played normal and *suicide* versions of games. In the suicide version, the rules are the same but the goals are inverted. For example, in Tic Tac Toe, the program aligning three marks looses the game. The playing time was 30 seconds and Ary was able to play about 100 games per second when choosing the first move. It did well, except on Connect Four Suicide were it lost its two games against ClunePlayer.

At the end of the day, the organizers sent a non-obfuscated game, and Ary crashed because we had not anticipated that atoms could contain the $+$ character.

On the sixth day, the programs played board games, including Blocker, used in the first GGP tournament and the classic Othello. In this last game, Ary made only 15 playouts in 30 seconds at the beginning of the game.

Fourth week

On the seventh day and eigth day, the programs played two single player games were the player must attain a target while eliminating or avoiding opponents. All other games were variations on classical games: Pentago, Amazon, Skirmish (a game played with chess pieces where the reward is propotionnal to the number of enemy pieces captured), and Checkers (without multiple captures and reward proportionnal to the number of captured pieces). On checkers, Ary made about 13 playout per seconds. The playing time for the games was usually 30 seconds.

At the end of the eigth day, which closed the qualifying phase Ary was back at the third place.

Discussion

The precise interpretation of the results of Ary during the tournament is a difficult task: games played in different matches differ vastly in complexity; bugs were corrected in Ary and probably in the other players as well; we know the moves of the other players only in the matches they played against Ary.

The absence of transposition tables was costly in some simple games: the 3×3 puzzle was unsolvable without them. Given the initial reflexion time of 10 seconds and the number of playout per second at the beginning of match, transposition tables would probably not be sufficient to solve the puzzle in the minimum number of moves. It is unclear if their use would have been sufficient to solve the puzzle at all.

The principal lessons that can be gained from the participation of Ary to the 2007 tournament is that Monte-Carlo is competitive; it gives decent results, even in games where the number of playout at the beginning of a match is so low that the choice of the first moves is actually nearly random. As the game progress to its conclusion, the number of playout increases and the move selection becomes better, compensating for the deficiencies at the beginning.

Replacing Pure Monte-Carlo by UCT

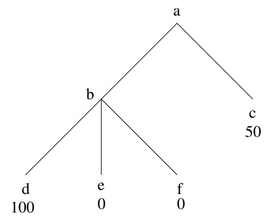
After the qualifying phase of the competition, we decided to replace the Pure Monte-Carlo algorithm used at the core of Ary move selection by an algorithm that is known to give good results for difficult games: the *Upper Confidence bounds applied to Trees*, usually named UCT (Kocsis & Szepesvàri 2006).

We will use *abstract move tree* to name the complete move tree as defined by the rules of the game and the current board status. In contrast, the *move tree* without qualification will denote the subset of the abstract move tree that the exploration algorithm has constructed.

The UCT algorithm

The basic idea of UCT is to add to Monte-Carlo explorations of the abstract move tree an informed way to choose the branches that will be explored. A move tree is constructed incrementally, with a new node added for each Monte-Carlo exploration. On the next exploration, a path is chosen in the already built move tree by choosing the branch whose gain is maximum, as estimated by the Monte-Carlo algorithm plus confidence in the estimation, calculated by a function of the number of explorations of the node t and of the number of exploration of the branch s as $\sqrt{log(t)/s}$. When arriving at a leaf node of the move tree, if it is not a terminal situation (i.e. it is not a leaf of the abstract move tree), then a new node is added to the tree and a Monte-Carlo simulation is started to obtain an evaluation of this node, also used to update the evaluation of the parent nodes.

Figure 1: A simple move tree for a single player game that PMC won't solve.



UCT has been applied with success to Monte-Carlo Go in the program MoGo (Gelly *et al.* 2006; Gelly & Silver 2007) among many others. UCT and its variations are very successful in the game of Go, and the current best Go programs use UCT.

We were particularly interested in the UCT algorithm because of its way to adapt itself to the state of exploration of the abstract tree: when little is known about the abstract move tree, UCT will choose unexplored branches. When all the branches from a node have been explored, UCT will tend to re-explore the most promising ones: this tendency is controlled by a constant C, that is used to multiply the confidence upper bound $\sqrt{log(t)/s}$. At the limit, when all the nodes of the abstract tree have been explored, UCT is favoring the better branches and will converge to the same choice as a mini-max exploration.

The next sub-section presents a very simple game where Pure Monte-Carlo will make a bad choice while UCT will find the good move.

A game that PMC won't solve and UCT will

Imagine a one player game where the player descends into a tree until a leaf where it finds its reward. The abstract move tree is represented in the figure 1; each node of the tree is named by a letter for reference and the gain for the player is indicated by a number between 0 and 100 in the leaf nodes. The game can be represented in GDL with the rule of the table 2

It is clear that the best sequence of moves for this game is to descend into the b and d nodes to find a reward of 100 but Pure Monte-Carlo is not able to discover this strategy: the descendants of b are explored about the same number of times, so the mean gain for b is estimated as 33.3, the mean of the rewards of the nodes d, e and f. It is the c node whose reward is 50 that is chosen.

In contrast, UCT will favor the exploration of the branch $b \rightarrow d$ over $b \rightarrow e$ and $b \rightarrow f$. As the number of explorations of this branch augments the estimated reward of b will tend to 100 and UCT will choose the winning move. The exact number of explorations necessary for UCT to make the correct choice depends of the exact value of the constant.

Table 2: A GDL representation of the game that PMC can't solve.

```
(ROLE player)
; representation of the tree
(edge a b) (edge a c) (edge b d) (edge b e)
(edge b f)
; rewards in leaf nodes
(leaf c 50) (leaf d 100) (leaf e 0)
(leaf f 0)
; starting situation
(INIT (current a))
; theorems
(<= (LEGAL (DOES player (goto ?y)))
    (TRUE (current ?x)) (edge ?x ?y))
(<= (NEXT (current ?x))
    (TRUE (DOES PLAYER (goto ?x)))
(<= TERMINAL (current ?x) (leaf ?x ?n))
(<= (GOAL player ?n)
    (TRUE (current ?x)) (leaf ?x ?n))
```

Discussion

Adding the UCT algorithm to Ary was relatively straightforward: it represented less than 1000 lines of C. Informal tests, done between the qualifying phase and the final competition let us think that it provides a significant advantage over the PMC algorithm used by Ary in the qualifying phase of the competition.

As a bonus, the use of UCT permitted to make useful computation during the opponent thinking time: Ary systematically explores the tree after each move is played, even when it has only one legal move. When the next moves of all the players are sent by the Game Master, they are used to choose a branch, whose node is set as the new root of the move tree. Ary thus re-use the accumulated information gathered in this sub-tree during the preceding explorations. Similarly, the initial thinking time is used to start building the game tree.

We tried to adjust the constant C used in UCT to multiply the interval of confidence on the basis of tests on simple games. As might be expected, the results were very dependent on the nature of the game: simple games whose whole move tree can be explored versus long games where it is only possible to explore a small part of the game. The number of calls to Monte-Carlo tends to diminish as the tree is explored, but tentatives to modify dynamically the constant during the game on this basis did not give good results and we finally used a constant of 50, as the reward varies between 0 and 100.

We had to adapt UCT to games with simultaneous play. A sound adaptation would have been to compute for each node a gain matrix from the mean of the previous explorations of that branch, to adjust the values of this gain matrix with the upper confidence bound and use it to select the branch to explore. We choosed a simpler solution, reminiscent of what had worked well in the qualifications for PMC: the moves are chosen independently for each player and these independent moves are combined to choose the next branch.

Participation in the final phase

Due to a combination of a crash of its usual machine and of our inability to properly read a schedule, Ary did not show up on the field for the first match of the final phase and thus was eliminated from the competition.

Conclusion

We have presented Ary, a program for General Game Playing based on Monte-Carlo, that participated in the qualifying phase of the 2007 General Game Playing tournament where it ranked third.

Future work with Ary go in three directions: make a better use of UCT, explore a bigger part of the move tree and symbolic manipulations on the rules of the game.

The algorithm UCT proved itself as a good algorithm for General Game Playing, but questions are still unsolved: it is inappropriate in the situations where the number of playouts is low. Preliminary explorations of variants mathods to choose the moves in these situations like *All Moves As First* did not give good results, but a finer adaptation may attain better results.

The constant used in UCT to balance between exploration and exploitation is, for the time being, fixed for the duration of the game. We intend to search criteria to adapt this constant to the portion of the move tree that is effectively explored.

We do not expect the correct adaptation of UCT to simultaneous games to bring a significant improvement over the games used in the 2007 competition, but this adaptation is nonetheless necessary as it is easy to design a game where the current simplification used in Ary induces poor play.

To have better results from UCT, it is necessary to maximize the number of playouts. This result with be obtained by a diminution of the time spent in the interface between the core of Ary and the Prolog interpreter, either by letting the interpreter play the whole playout or by replacing the Prolog interpreter by an ad'hoc inference engine. We also intend to add transposition tables and consider parallelizing the move tree explorations.

At last, there are manipulation on the rules that can be adapted to be of use without modifying the structure of the program, like the recognition and exploitation of the existing board symmetries.

References

Berlekamp, E.; Conway, J. H.; and Guy, R. K. 1982. *Winning Ways*. Academic Press.

Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*, 1134–1139.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *ICML*, 273–280.

Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modification of UCT with patterns in monte-carlo go. Technical Report 6062, INRIA.

Kocsis, L., and Szepesvàri, C. 2006. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *AAAI*, 1191–1196.

Wielemaker, J. 2003. An overview of the SWI-Prolog programming environment. In Mesnard, F., and Serebenik, A., eds., *Proceedings of the 13th International Workshop on Logic Programming Environments*, 1–16. Heverlee, Belgium: Katholieke Universiteit Leuven. CW 371.