Monte-Carlo Tree Search for General Game Playing

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 present a game engine for general game playing based on UCT, a combination of Monte-Carlo and tree search. The resulting program is named ARY. Despite the modest number of random games played by ARY before choosing a move, it scored quite well in the qualifying phase of the annual general game playing tournament hosted by AAAI.

1 Introduction

Programs to play games are usually based on a restricted number of features of a game, identified by experts of that game, that are exploited by the programmer to try to obtain a good level of play.

For game playing, a number of strategies are known and one chooses among them the ones that will give the best level of play.

General Game Playing is a special variant of game playing, usually played by programs, where the ability to automatically extract important features of a given type of game is stressed : the players are given the rules of the game, these rules are analyzed for a given period of time, and then the game is played with the usual time constraints.

The application of Monte-Carlo tree search to General Game Playing is presented in this paper. The second section presents Monte-Carlo tree search. The third section deals with General Game Playing. The fourth section details the representation of a game. The fifth section is about the Monte-Carlo algorithms we have tested. The sixth section gives experimental results.

2 Monte-Carlo tree search

Monte-Carlo tree search is a general technique that has proved useful in computer Go. Monte-Carlo Go can be traced back to GOBBLE [1]. An efficient combination with global search has been proposed by Rémi Coulom in his program CRAZY STONE [2]. It consists in adding a leaf to the tree for each simulation. The choice of the move to develop in the tree depends on the comparison of the results of the previous simulations that went through this node, and of the results of the simulations that went through its sibling nodes.

The UCT algorithm [3] uses a similar tree development, but uses another formula to select the moves to develop. UCT consists in exploring the move that maximizes

$\mu_i + C \times \sqrt{log(t)/s}$. The mean result of the games that start with the c_i move is μ_i, the number of games played in the current node is t, and the number of games that start with move c_i is s. The C constant can be used to adjust the level of exploration of the algorithm. High values favor exploration and low values favor exploitation.

UCT has been applied with success to Monte-Carlo Go in the program MOGO [4, 5] among many others. UCT and its variations are very successful in the game of Go, and the current best Go programs use UCT.

Monte-Carlo tree search can also be applied to other games, but as it is a recent technique it has not yet been tested on many other games.

3 General Game Playing

General game playing is a longstanding goal of Artificial Intelligence [6]. Realization of general game playing program can be traced back to the seminal work of J. Pitrat on GENEJEU [7].

Other works that explicitly use first order logic to represent the rules of different games and reason on them so as to produce players are METAGAME for chess-like games [8] and INTROSPECT [9].

Recent work also includes the development of general game players for a broader class of games, including board and card games [10].

In 2005, 2006 and 2007, the Stanford logic group organized the General Game Playing competitions at the AAAI conferences. Games are represented using the Game Description Language, based on first order logic.

The winners of the 2005 and 2006 competitions used classical tree search based on Alpha-Beta [11, 12]. Cadia, the winner of the 2007 competition uses the UCT algorithm.

Ignoring the rules of the game before the match begins does not allow giving game specific knowledge to the game playing programs. The programs have to use general algorithms or recognize the game type after the rules and deduce the best algorithm. Progress in General Game Playing is likely to produce more general algorithms than progress in specific games.

4 Representation of a Game

This section deals with the rules of the games used in GGP. The first subsection details the representation given to the players. The second subsection explains the internal representation used in ARY. The third subsection shows the interface to the Prolog interpreter.

4.1 Representation of Games Rules with First Order Logic

In General Game Playing, a game is represented by a set a first order logic expressions representing the rules and three parameters : the *role* of the program, the time before the first move (*initial reflection time*) and the time between moves. In the 2007 competition, these times varied between 10 seconds and 20 minutes.

The rules of the game are expressed with a few keywords: `init` indicates that the following expression represents a feature of the board at the beginning of the game, `does` is used for representing a move, `legal` for describing a legal move, `next` for the state of the board after a move is played, `terminal` is true when the game is finished, and `goal` permits to know the score of each player at the end of the game.

Axioms and theorems are communicated in a syntax reminiscent of Lisp forms, with a '?' indicating variables. For example the expression

```
(<= (LEGAL (DOES ?player (drop ?x ?y)))
    (true (empty ?x ?y)) (true (active ?player)))
```

might be used to indicate that the active player can drop a piece on any empty board cell, while:

```
(<= (NEXT (occupied ?x ?y ?player))
    (DOES ?player (drop ?x ?y)))
```

describes a feature of the board after a move.

4.2 Internal representation of KIF expressions in ARY

All the expressions regarding the game description and status are stored internally as lists implemented like Lisp expressions. Unused cons cells are added to the free list explicitly by the code: there is no garbage collection.

A hash code is stored with each cons cell. It identifies the contents of the list. The hash code is calculated like a Zobrist hash based on 64 bits random numbers attributed to atoms on creation. This hash code is used in hash tables that are the primary data structures for storing collections of lists.

Two tables contain all the expressions describing moves and board states encountered from the start of the game. It avoids the memory consumption due to similarities between different nodes. Except in some of the games specifically conceived to stress the program robustness, these tables always keep a small size fitting easily in memory.

4.3 Conversion to Prolog

A Prolog interpreter is used as an inference engine, with a clear cut interface with the program. Most of the data manipulation is done in KIF; the conversion between Prolog and KIF is part of the interface and consists in about 450 lines of C code, including comments and self test support.

After receiving the description of the game to be played from the Game Server, the theorems are loaded once in the interpreter. The initial situation, described by assertions featuring the INIT keyword, is also loaded as ordinary facts.

To enumerate the legal moves in the current position, for each *player* the Prolog interpreter is repeatedly given the goal `legal(`*`player`*`, Move)` until failure. The answers are immediately converted to KIF and stored in the hash table of legal moves for each player. For example, Prolog replying to `legal(playera, Move)` that `Move`

`= pass)` is translated as the KIF expression `(DOES PLAYER-A PASS)` that will be used to play the move.

Similarly, a terminal position is detected trying to prove the `terminal()` goal, and the score of a player by asking `score(player, Score)`

The current status of the game in the Prolog interpreter is updated by retracting and asserting facts. This update is done incrementally: facts that do not change are not reloaded. This is particularly efficient in the many games where a single move only affects the local situation of the board. For example, in Tic Tac Toe the board is described by nine assertions; after a move, only one assertion has to be modified to reflect the modification of the board state.

All the interface between the search algorithm and the Prolog interpreter is in a single file of 600 lines of C code, including comments and support for self test.

A few adjustments were necessary in the KIF description of the game to avoid problems with the Prolog interpreter. All the atoms appearing after an opening parenthesis in a KIF expression are declared as predicate with the correct arity in the Prolog interpreter, by asserting and then retracting an arbitrary expression. The expressions in the right part of the theorem are reordonated to let the ones without variables come first.

5 Monte-Carlo Tree Search

We present here the different Monte-Carlo algorithms we have tested. All are based on playouts where random games are played until reaching a terminal situation, whose evaluation is used to qualify the moves used.

The first subsection describes the most simple, pure Monte-Carlo that was used in the 2007 competition. The second subsection deals with UCT, and the third with the *All Moves As First* variant.

5.1 Pure Monte-Carlo

In pure Monte-Carlo, all the moves of all the playouts are chosen randomly and the final score of each playout is used to qualify its first move. When time is elapsed, the move with the greatest mean is played.

This algorithm has an advantage in its simplicity and its robustness. It has a small memory footprint as nothing has to be stored permanently, except the legal moves from the beginning of the game.

5.2 UCT

For UCT, when a move has not been played in the tree, it is tried before already explored moves. When all moves are explored, UCT chooses the move that maximizes $\mu_i + C \times \sqrt{log(t)/s}$ for each player. Scores of the games are in the range 0–100, so we use $C = 50$. For each playout, the move tree is expanded by one node. This way, the following playouts can descend into the tree without replaying the move, which is a slow operation in our implementation.

In games where players move simultaneously, we use a simplification for the sake of simplicity and rapidity. A score is computed for each move of a player by summing the score of the nodes where it appears. The move for each player is then chosen independently.

No attempt is done to make a difference between zero sum and collaborative games. Each player's move is chosen on the basis of this player's score, ignoring the score of the other players.

5.3 All Moves As First

For *All Moves As First* (AMAF), the result of a playout is used to qualify all the moves played in this playout.

The first moves of the playouts are chosen by descending a tree built as in UCT, using the mean of the corresponding moves. Once a leaf is attained, the tree is also expanded by one node and a random playout is played up to the end of the game.

The legal move at the root with the greatest mean is played when time is elapsed.

6 Experimental Results

This section contains a brief description of the participation of ARY in the qualifying phase of the 2007 competition. Then we compare the performance of the three algorithms Pure Monte-Carlo, UCT and AMAF on different games from the last two weeks of the 2007 competition qualification phase.

6.1 Results in Competition

The qualification phase of the 2007 AAAI tournament consisted in four weeks of tournaments, with two days of play for each week. Each day, the programs played about ten matches in different games.

The first week, the program played games in order to test the basic functionality on simple games. The second week, the games played stressed the limits of the programs, testing large number of possible moves, large number of rules or large state spaces. The third and fourth week, the programs played variations on classical games including Othello, simplified Chess, Checkers and Amazons.

ARY played 83 matches of approximately 35 different games. The algorithm used was pure Monte-Carlo. Among the eight participants, ARY ranked third of the 2007 GGP qualification rounds.

Naturally, the mean number of random games played by ARY varied enormously, depending on the nature of the game and the phase of the match. To eliminate this last bias, we consider only the number of random moves played before playing the first move of each match.

On the whole competition, the mean number of random games is very close to 150 games per second. On a simple game like Tic Tac Toe, the figures are about 4500 games per second. On longer games like Othello, they can drop to 2 seconds for one complete random game, and even 5 seconds for one version of Amazon where the number of moves is very high. When considering only the more interesting games of the third and fourth week, the mean number of random games per second is approximately 65 per second.

6.2 Result in self-play

This section presents results obtained by making ARY play against itself on different games used in the last two weeks of the GGP 2007 qualifying phase : Quarto, Breakthrough with normal and inverted goals, Pentago with normal and inverted goals Othello and Amazon. Another version of Amazon could not be used for testing as its large number of moves per node induced a memory exhaustion and a program crash for UCT, due to a default in the implementation of the move tree.

As all the results obtained by self test, the results of these experiments have to be considered *cum grano salis*. Some of the algorithms may present deficiencies that the variant of the same player is not able to exploit, but that would be disastrous against another opponent.

UCT vs Pure Monte-Carlo Here we compare the scores of ARY against itself, when using Pure Monte Carlo against UCT. Each set of rules was tested in twenty games, alternating the first and the second player. The results are presented in the table 1.

Table 1. UCT vs Pure Monte-Carlo

game	first player	UCT	PMC
Quarto	UCT	60	40
	PMC	75	25
Breakthrough	UCT	90	10
	PMC	60	40
Breakthrough (Suicide)	UCT	90	10
	PMC	80	20
Pentago	UCT	60	40
	PMC	80	20
Pentago (Suicide)	UCT	60	40
	PMC	90	10
Othello	UCT	75	25
	PMC	90	10
Amazons	UCT	80	20
	PMC	70	30
Checkers	UCT	96.4	88.4
	PMC	98.2	86.4
Mean		78.41	32.18

As can be seen from the table, UCT allows a significant gain is strength. It wins each set of ten games, playing as the first or the second player.

A part of the advantage of UCT over Pure Monte Carlo is probably due to the tree constructed by UCT that allows to play the first moves of a playout by a simple tree descent, with no need to generate the legal moves and modify the board state, thus avoiding the slow interaction with the Prolog interpreter.

Note that Checkers is not scored as a zero sum game but according to the captured material.

UCT vs *All Moves As First* In the same vein as the preceding section, we present here the results of ARY using UCT playing against itself using *All Moves As First*. The score presented are the means of the scores obtained in twenty games, alternating first and second role. The results are presented in table 2.

Table 2. UCT vs *All Moves As First*

game	first player	UCT	AMAF
Quarto	UCT	60	40
	AMAF	95	5
Breakthrough	UCT	80	20
	AMAF	50	50
Breakthrough (Suicide)	UCT	70	30
	AMAF	70	30
Pentago	UCT	50	50
	AMAF	30	70
Pentago (Suicide)	UCT	75	25
	AMAF	40	60
Othello	UCT	40	60
	AMAF	40	60
Amazons	UCT	40	60
	AMAF	75	25
Checkers	UCT	100	65
	AMAF	100	66.6
Mean		63.44	44.79

On some games, *All Moves As First* obtains better results than UCT, probably because, for these games, the number of playouts is so small that the information, gathered by the statistics on the first moves only, is not relevant. On the other hand, the advantage of UCT in the last phase of the game compensates for its disadvantage in the beginning in most of the studied games.

7 Conclusion and Future Research

UCT is a competitive algorithm for General Game Playing. It outperforms Pure Monte-Carlo on seven games from the 2007 AAAI competition. Results of competition between UCT and *All Moves As First* are more balanced and seem to be game dependent.

The time given to the program to analyze games before playing could be used to match UCT and *All Moves As First* to decide which one to use, and in which phase of the game.

A version of ARY using Pure Monte-Carlo ranked 3 out of 8 in the 2007 competition, and we believe the current UCT-based version would have scored even better.

To avoid the time consumed by translating question and answer between the core of the algorithms and the Prolog interpreter, we intend to integrate the program and the logic engine, either by writing a unifier in C or by deporting at least the Monte-Carlo exploration into the Prolog interpreter.

We will parallelize the search to exploit the capacities of SMP and multi-core processors. Stochastic algorithms are good candidates for parallelization, as the branches can be explored independently, with few interactions.

References

1. Bruegmann, B.: Monte Carlo Go. white paper (1993)
2. Coulom, R.: Efficient selectivity and back-up operators in monte-carlo tree search. In: Computers and Games 2006. Volume 4630 of LNCS, Torino, Italy, Springer (2006) 72–83
3. Kocsis, L., Szepesvàri, C.: Bandit based monte-carlo planning. In: ECML. Volume 4212 of Lecture Notes in Computer Science., Springer (2006) 282–293
4. Gelly, S., Wang, Y., Munos, R., Teytaud, O.: Modification of UCT with patterns in monte-carlo go. Technical Report 6062, INRIA (2006)
5. Gelly, S., Silver, D.: Combining online and offline knowledge in UCT. In: ICML. (2007) 273–280
6. Pitrat, J.: Games: The next challenge. ICCA Journal **21**(3) (1998) 147–156
7. Pitrat, J.: Realization of a general game-playing program. In: IFIP Congress (2). (1968) 1570–1574
8. Pell, B.: A strategic metagame player for general chess-like games. In: AAAI. (1994) 1378–1385
9. Cazenave, T.: Système d'Apprentissage Par Auto-Observation. Application au jeu de Go. Phd thesis, Université Paris 6 (1996)
10. Quenault, M., Cazenave, T.: Extended general gaming model. In: CGW 2007. (2007) 195–204
11. Clune, J.: Heuristic evaluation functions for general game playing. In: AAAI. (2007) 1134–1139
12. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: AAAI. (2007) 1191–1196