Ary: A Program for General Game Playing

Jean Méhat, Tristan Cazenave

Université de Paris 8, Saint Denis, 93536 Cedex, France
email: {jm,cazenave}@ai.univ-paris8.fr

General Game Playing

General Game Playing consists in building programs to play the games they have never met before. A Game Master sends the rules to the participants and after a delay devoted to game analysis, the programs start playing. It can be applied to games with any number of players, with alternative or simultaneous moves, for zero-sum or collaborative games types.

General Game Playing avoids the shortcomings of current specialized game playing programs that cannot adapt to other domains than the game they were programmed for. At the very least it induces a broad exploration of the characteristics of decision situation to identify automatically the heuristics that may give good results in this situation.

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. The following figure contains a representation in GDL of a binary version of the simultaneous play game *My father has more money than yours* [Berlekamp, Conway, & Guy1982]. Keywords of GDL are written in upper case.

```
(ROLE left) (ROLE right)
(LEGAL (DOES ?player (tell 0))
(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)))
```

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.

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.

Ary uses Prolog for rule interpretation

We made the choice to use a Prolog Interpreter as an inference engine for the interpretation of the rules of the game.

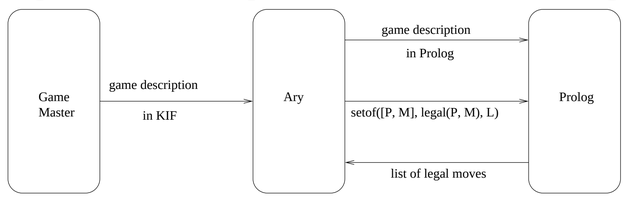

Figure 1: Ary translates the game description into Prolog and loads it into the interpreter. The characteristics of a position are then obtained through the Prolog interpret.

The translation process is relatively straightforward, with slight modifications to accomodate Prolog and GDL differences, and reordonnancement of the clauses to try to attain better performances.

The game theorems are loaded into the interpreter once or all. Board status are then loaded via calls to `assert` and `retract` to modify the view of the current situation in the interpret. The transition from one status to another is done incrementally, asserting and retracting only the changed properties.

Monte-Carlo explorations

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.

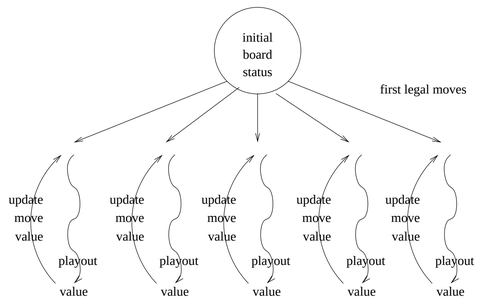

Figure 2: Initial moves are evaluated by random playouts.

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.

UCT tree construction

Ary uses *Upper Confidence bounds applied to Trees*, usually called UCT. UCT adds to Monte-Carlo explorations of the games move tree an informed way to choose the branches that will be explored. A subset of the 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 [Kocsis & Szepesvàri2006].

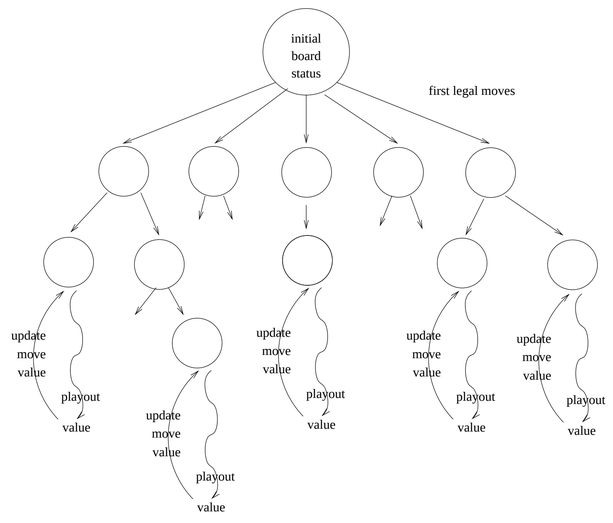

Figure 3: UCT combines the construction of a tree of moves with Montecarlo exploration to approximate a value of a node.

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 use a simpler solution, reminiscent of what worked well in the qualifications of 2007: the moves are chosen independently for each player and these independent moves are combined to choose the next branch.

UCT, Montecarlo and Minimax

A clear advantage of UCT over a Pure Monte Carlo exploration is that it is garanteed to converge to the same value as a Minimax exploration. For example in the game whose move tree is represented on figure 4, Montecarlo will evaluate move b as an average of the value of the leaves d, e and f, and will choose to move at c.

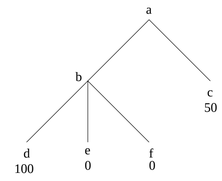

Figure 4: A simple move tree for a single player game that Pure Montecarlo won't solve.

UCT on the other hand will favor the exploration of the branch d over e and f and the value of node b will converge to 100, while these of c will stay at 50.

Exploration, Exploitation and Refutation

UCT can be tuned to explore, by multiplying the confidence bound by a large factor. In limited time explorations of move trees, this lead to poor performances it is unable to find easy refutations of winning sequences of moves found by a random playout.

On the other hand, when UCT is tuned to keep searching in the promising moves sub-trees, it explores only superficially branch that may contain winning moves.

The current version of Ary tries to adapt the confidence bound to the characteristics of a node.

Implementation of Ary

Ary is mainly written in C. It amounts to approximately 10K lines of code. The current version use either SWI-Prolog or YAP; debugging is usually easier with SWI, but YAP gives better performances [Wielemaker2003], [Costa *et al.*2000].

It uses transposition tables. A version uses parallel computing based on MPI for the playouts, but it is not sufficiently tested for use in the competition.

Ary results in the GGP AAAI Tournaments

In the qualifying phase of the 2007 competition, a preliminary version of Ary using only Montecarlo went out at the third place.

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 2007 final phase and thus was eliminated from the competition.

The current version described here rated third in the 2008 qualifying phase.

References

[Berlekamp, Conway, & Guy1982] Berlekamp, E.; Conway, J. H.; and Guy, R. K. 1982. *Winning Ways*. Academic Press.

[Costa *et al.*2000] Costa, V. S.; Damas, L.; Reis, R.; and Azevedo, R. 2000. The Yap prolog users manual, 2000. Technical report.

[Kocsis & Szepesvàri2006] Kocsis, L., and Szepesvàri, C. 2006. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

[Wielemaker2003] 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.