The Monte-Carlo Approach in Amazons

Julien Kloetzer[1], Hiroyuki Iida[1], and Bruno Bouzy[2]

[1] Research Unit for Computers and Games
Japan Advanced Institute of Science and Technology
[2] Centre de Recherche en Informatique de Paris 5
Université René Descartes

Abstract. The game of the Amazons is a quite new game whose rules stand between the game of Go and Chess. Its main difficulty in terms of game programming is the huge branching factor. Monte-Carlo is a method used in game programming which allows us to overcome easily this problem. This paper presents how the Monte-Carlo method can be best adapted to Amazon programming to obtain a good level program, and improvements that can be added to it.

1 Introduction

The game of Amazons (in Spanish, *El Juego de las Amazonas*) has been invented in 1988 by Walter Zamkauskas of Argentina. Although it is very young, it is already considered as being a difficult game: its complexity is between Chess and the game of Go.

The main difficulty of the game of Amazons is its complexity: the average number of moves is 80, with a branching factor of approximately 500, and no more than 2176 available moves for the first player. With this in head, we can clearly see that an exhaustive full tree-search is a difficult task: there are no more than 4 millions different positions after two moves. Moreover, even if many of theses moves are clearly bad, there are often positions where more than 20 moves can be considered as "good" moves, and selecting these moves is a hard task [2].

Monte-Carlo (short: MC) is a simple game-independent algorithm which has recently proven to be competitive for the game of Go, especially including a part of Tree Search (different from the classical minimax approach), and knowledge of the game. Now, considering that the game of Go has the same main drawback as the Amazons, a huge branching factor, and that both these games are territory-based games, thus increasing the similarity, we can expect the Monte-Carlo approach to give good results for the game of Amazons.

After a brief description of the rules of the game, section 2 of this paper focuses on the main core of a Monte-Carlo Amazons program. Section 3 presents some improvements that can be added to it, with the results discussed in section 4. Section 5 focuses on our latest improvement to the program, before concluding and discussing future works in section 6.

2 Using Monte-Carlo for the game of Amazons

The rules of this game (usually called simply "Amazons") are very simple: it is played on a square board of 10x10, sometimes less but this size is the classical one. Each player begins with 4 Amazons, placed all over the board (left of figure 1). A player move consists first on moving one of his Amazons: it can be moved in every direction in a straight line, on any square accessible from the Amazon, exactly as Queen in Chess. Then, after having moved an Amazon, the player chooses a square accessible from the one on which his Amazon just landed, and *shoots an arrow* on this square: this square becomes blocked until the end of the game. No further Amazon move or arrow shot can go through it or land on it (right of figure 1).

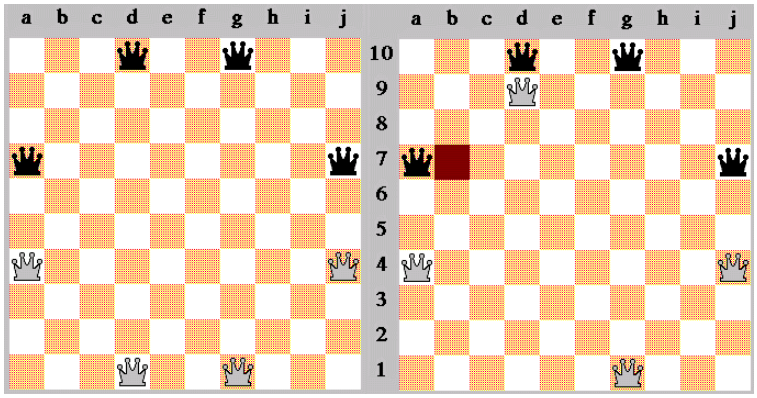

Fig. 1. Beginning position in Amazons (left) and after one move (right).

Each player alternatively makes one of these moves, so a square is blocked on each move. The first player that cannot move any more loses the game, and the score of the game is usually determined as being the number of moves that the other player could have played after it.

All Monte-Carlo Amazons programs should include recent developments made to combine MC and Tree-Search (short: TS). A pseudo code is given in figure 2.

One run (or playout) of the evaluation consists on three steps:

- First, after the search tree used in the evaluation has been initialized to the root itself only (line 2), a new node of the game tree is selected to be added later to the current search tree (line 6), combined with its evaluation.
- Then, a completely random game is performed, starting from the given node, until the end of the game (line 5). The evaluation given is usually either the score of the game (if available), or the information win/loss/draw.

```
 1. Function find_move ( position )
 2. treeSearch_tree = tree ( position )
 3. while ( time is remaining )
 4.   Node new_node = search_new_node ( treeSearch_tree )
 5.   v = evaluate ( new_node )
 6.   treeSearch.add ( v, new_node )
 7. return best_move()
 8.
 9. Function search_new_node ( tree )
10. node = root ( tree )
11. while ( number_unvisited_children ( node ) != 0 )
12.   node = find_best_children ( node )
13. node = random_unvisited_children ( node )
14. return node
```

Fig. 2. Pseudo-code for Monte-Carlo using tree-search.

– Finally, the given node is added to the current search tree: the evaluation given is stored in all the nodes that have been visited in the function search_new_node and in the new node.

The search function is the main core of the tree-search Monte-Carlo model. In previous versions of MC, it always returned one of the children node of the root, either chosen randomly, or so that each node is visited the same number of time [1], or chosen by some knowledge [4]. Following the algorithm UCT [7], our program visits the current search tree by exploring nodes that maximizes the score given by the formula:

$$\text{Score(node(i))} = \text{Evaluation(node(i))} + \text{C} * \sqrt{\frac{ln(nbVisits(parent(node(i))))}{nbVisits(node(i))}} \quad (1)$$

The first term of formula (1), the evaluation, is usually given either by the expected win/loss ratio of the node, or the expected average score. The number of visits of one node is the number of timeit was selected in the process of searching for a new node (line 9 of figure 2). Both these values (number of visits and average evaluation) are updated after each playout (line 6). The second term allows nodes not to be forgotten during the process, raising if the node is not visited while his parent is. Finally, the factor C has to be tuned experimentally.

In the field of Amazons playing, some features need to be discussed:

– Each move consists of two actions (Amazon movement + arrow shot), and usually, one node results from a combination of these actions. However, we can also choose to split every action decision in two: in every random game, an Amazon movement is selected at random, and then an arrow shot from this Amazon, not a combination. Also, we can do the same with the search

tree used by UCT, by not using a usual two levels tree, but a four levels one. On the first level are the positions obtained after an Amazon movement, on the second the positions obtained after an arrow shot from the Amazon just moved, and similarly for the third and fourth level, with movements and shot from the other player.
This changes should allows us basically to run more playouts, and thus to increase in a very simple way the level of the program, because it does not have to compute every move at each position in the random games.
- The evaluation has also to be chosen accordingly to the game which is played. A Win-Loss ratio is usually used for MC + TS in game programming [6], but we could also use an average score, or a combination of both.

Tests and discussion of these three features (splitting in the random games, splitting in the tree used by UCT and the evaluation) are given in section 4.

3 Improving the rand games

Our program (Campya) includes the algorithm presented in section 2 to choose a move. However, at this state, it lacks seriously of some knowledge of the game, and can easily be defeated by a fair human player.

Improvements to a Monte-Carlo with Tree-search program can basically be made at three levels:

- In the random games, by adding knowledge to it [3]
- In the tree-search part, changing the behaviour of UCT or using other techniques [5]
- At the final decision of the move, for example by pruning moves [4]

Improving the random games has already proven to be a good way to improve the level of a Monte-Carlo program, by adding knowledge to create pseudo-random games. Moves can be chosen more or less frequently according to patterns or to simple rules, as we did here for the game of Amazons. We decided to focus on this method to improve the level of Campya.

3.1 The liberty rule

Mobility is an important factor in Amazons. Having an Amazon which is completely or almost completely enclosed at the beginning of the game is like fighting at 3 against 4 for the remaining game. We defined the number of liberties of an Amazon as the number of empty squares adjacent to this Amazon, using a concept similar to the game of Go. Then, we added the following rules to the random games:

- Any Amazon with 1 or 2 liberties has to be moved immediately
- Any opponent's Amazon with 1 or 2 liberties should be enclosed if possible

Two liberties is a critical number: if they are adjacent, one can move an Amazon on one of these, and shoot an arrow on the other one. This way, we punish bad moves, and try to avoid being punished.

3.2 Pruning moves from enclosed Amazons

We say that an Amazon is isolated if any of the squares accessible from this Amazon in any number of moves cannot be accessed by opponent's Amazons. An isolated Amazon is inside a territory and should not be moved, except in situations of Zugzwang. Since this concept is way beyond the simplicity we search in the random games, we added the following rule to the random games:

– Any isolated Amazon should not be moved if possible

Obviously, if all Amazons are isolated, one has to be moved. But in this case, the game should be considered to be over: no player can now access to its opponent territory.

Tests and discussion of these two features will be discussed in section 4.

4 Experiments and results

Due to the absence of popular Amazons servers and game protocol, testing of these improvements has been realised through self-play games against a standard version of our program. Each test consisted of runs of 250 games with 3 minutes per player, each player playing half of the time as first player. Some games were also played by hand against an other Amazons program: Invader [9], with an equivalent 30 sec/move time setting for both programs.

The standard version used for testing, later called Vanilla Campya, uses a light version of the algorithm presented in section 2: Monte-Carlo without Tree-Search. Moves in the random games are split, and the evaluation of the games is given by their score. The results of the game-independent features are shown in table 1, and those of game-dependent features (liberty rule and pruning isolated Amazons moves) in table 2. Each feature or set of features was added to Vanilla Campya to produce a new version of the program, and then tested against the Vanilla version.

Table 1. Results of Campya integrating some features against a standard MC version.

Feature tested	Win ratio
Not splitting moves in the random games	20,4%
Evaluation by Win-Loss ratio	43,6%
Evaluation by combining score and Win-Loss ratio	63,5%
Using tree-search	81,6%
Using tree-search and combined evaluation	89,2%
(1) Using tree-search, combined evaluation, and splitting moves in the tree-search	96%

The results obtained by the version not splitting moves in the random games are conform to our intuition, with only 20% of win against the Vanilla version.

Further tests (not included here) showed us also that, even with an equivalent number of playouts, the non-splitting version was behind. The results obtained using different evaluations are a bit more surprising: it seems that, for the game of Amazons, evaluation with a Win-Loss ratio is not the key, and that the score alone is not sufficient either. Finally, the results obtained by integrating Tree-Search are not surprising: a Tree-Search based version of Campya is way above the others. Also, splitting the moves in the tree used by this version seems really effective, and not only because of the higher number of random games that Campya could launch: as for splitting moves in the random games, even with an equivalent number of playouts, the non-splitting version was behind.

Table 2. Results of Campya integrating some features against a standard MC+TS version.

Feature tested	Win ratio
Version (1) + liberty rule	94,4%
Version (1) + pruning	92,8%
Version (1) + pruning + liberty rule	96,4%

Version (1) in table 1 was used as a basis to test the knowledge-based improvements (liberty and pruning). Results obtained using them do not show a significant difference with the results given in table 1. However, considering that:

- Adding this form of external knowledge slowed the playouts, and thus did not permit us to launch as many as without, and
- The knowledge of liberties is especially useful against players who know how to exploit it, so not against Vanilla Campya,

we can still consider this integration of Amazons-knowledge in Campya as being an improvement. Moreover, the first few games played against Invader were terrific in the opening, because Campya had no knowledge of Amazon imprisonment. Adding it permitted our program to perform better games against Invader.

5 Integrating the accessibility in Campya

At this point, Campya still lacked an important knowledge, used a lot in other programs. The accessibility to a square by a player can be defined as the minimum number of Amazon move a player has to perform to place an Amazon on this square. This number is set to infinite if the player has no access to the square. It is used by many programs, such as Amazong [8]

Accessibility is territory-related: if a player has a better accessibility to a square than his opponent, this square has a higher chance to be part of this player's territory at the end of the game than to be part of his opponent's. This goes even more true as the game reaches its end. Lacking this knowledge,

our program could not understand the idea of potential territory, and thus was mostly crushed in the opening and middle game by other programs or good human players.

Integrating this knowledge in a Monte-Carlo architecture cannot be done easily: it requires lots of computations, and thus slows down the speed of the random games too much to be useful. However, it can be integrated as a new evaluation, which led us to this:

- random games are not any more evaluated by their a combination score + win/loss at the end of the game, but by the estimated score of the game after a fixed number of plies

Tests of this new feature have been done the same way as presented in section 4, with the difference that the reference version was not VanIlla-Campya any more, but the version (1) of table 1. Results are shown in table 3.

Table 3. Results of Campya integrating accessibility-based evaluation, in number of games (average score).

Version (1)	Version (1) + accessibility evaluation
45 (3)	205 (12)

Using the accessibility heuristic as a score evaluator allowed Campya to perform much better results, having more than 80% of win against its previous best version and losses by only a few points, any feature similar except for the evaluation. The games played against Invader also showed us that its level increased and that it was now able to understand the concept of potential territory, still not being able, in its actual version, to perform better than Invader, but showing lots of potential.

6 Conclusion and future works

We presented in this paper how to best integrate the Monte-Carlo method for the purpose of obtaining a good Amazons playing program. We discussed the main features, and proposed forcing moves by the liberty rule and pruning useless moves as ways to improve the level of the play. Finally, we proposed combining MC and an evaluation function as being the best way to obtain a good level program, thus using MC to explore a game tree and not any more to create an evaluation function.

Further works mostly include finding other light forms of knowledge to improve the random plays, specifically related to the endgame and the opening. Also, we would like to find the best way to combine Monte-Carlo tree-search and en evaluation function for the game of Amazons.

Acknowledgements

The picture of the Amazons boards in section 2 comes from the website www.solitairelaboratory.com

References

1. Bruce Abramson (1993), Expected-outcome: a general model of static evaluation, IEEE transactions on pattern analysis and machine intelligence 12:22, 182-193.
2. Henry Avetisyan, Richard J. Lorentz (2002), Selective search in an Amazons program, Computers and Games 2002: 123-141.
3. Bruno Bouzy (2005), Associating knowledge and Monte Carlo approaches within a go program, Information Sciences, 175(4):247257, November 2005.
4. Bruno Bouzy (2005), Move Pruning Techniques for Monte-Carlo Go, 11th Advances in Computer Game conference, Taipei 2005.
5. Remi Coulom (2006), Efficient Selectivity and Backup operators in Monte-Carlo Tree-Search, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy, 2006.
6. Sylvain Gelly, Yizao Wang, Remi Munos, Olivier Teytaud, Modifications of UCT with Patterns in Monte-Carlo Go, Technical Report 6062, INRIA
7. Levente Kocsis, Csaba Szepesvari (2006), Bandit based Monte-Carlo planning, 5th European Conference on Machine Learning (ECML), Pisa, Italy, Pages 282-293, September 2006.
8. Jens Lieberum (2005), An evaluation function for the game of Amazons Theoretical computer science 349, 230-244, Elsevier, 2005.
9. Richard J. Lorentz, Invader, http://www.csun.edu/ lorentz/amazon.htm
10. Martin Muller, Theodore Tegos (2001), Experiments in Computer Amazons, in R.J. Nowakowski editor, More games of No Chance, Cambridge University Press, 2001, 243-260