A Comparative Study of Solvers in Amazons Endgames

Julien Kloetzer, Hiroyuki Iida, and Bruno Bouzy

Abstract— **The game of Amazons is a fairly young member of the class of territory-games. The best Amazons programs play now at a high level, but can still be defeated by humans expert of the game. Our focus here is on the solving of endgames, with the goal of improving the general playing level of Amazons programs. Our comparative study of four solvers, DFPN, WPNS, Alpha/Beta and Monte-Carlo Tree-Search, shows that usual game-playing methods used for Amazons, namely Alpha-Beta and Monte-Carlo Tree-Search, are best suited for this task. This result also emphasizes the need of an evaluation function for the game of Amazons.**

I. Introduction

Recently, focus in game-programming has derived a lot onto territory-based games (mostly the game of Go, but also the game of the Amazons) and Monte-Carlo methods, and many very strong programs have emerged, achieving results unimaginable some years ago [1]. But these works often focus on making a very strong game-program able to play well during a whole game; few worked on improving some specific parts of the game [2].

However, for the purpose of building a strong strategy game program, several tasks are key to increase its level. It is usually not possible to build a strong program that just "plays well" in the whole game: plays well in the opening, plays optimally in the endgame, these tasks are very important points to consider to beat top human players.

In this paper, we study endgame solvings of one territory-based games, the game of Amazons. Traditional solvers (DFPN, Alpha/Beta) are considered, as well as some newer techniques such as Monte-Carlo and WPNS. The main goal still being to build a strong game-playing engine, we focus here on realistic endgames and solving in limited time.

We present in section II the game of the Amazons, as well as the different solving techniques considered in this paper. Section III presents a way of building a problem database and deals with the problem of comparing different sort of solvers, with results being presented and discussed in section IV. The conclusion follows in section V.

II. Background works

A. The game of the Amazons

The game of Amazons, also called Amazons, is a two-player deterministic game with perfect information. It was created in 1988 by Walter Zamkauskas. It is played on a 10×10 board, each player controlling 4 Amazons, moving like Queens in Chess (any number of squares in any direction). In turn, players move one of their Amazons, then "shoot an arrow" from the landing square of the Amazons, in any direction, any number of squares away (see figure 1). From now on, the square on which the arrow lands is blocked: no Amazons can move over or stop on, as well as further arrows. Each turn, the number of empty squares of the board is reduced by one by a shot, and the first player unable to move loses the game. At this moment, it is usually agreed that the score of his opponent is the number of moves he can still make after the pass.

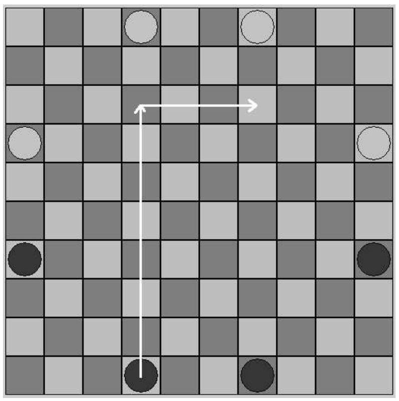
Fig. 1. Classical Amazons starting position with a possible first move and shot

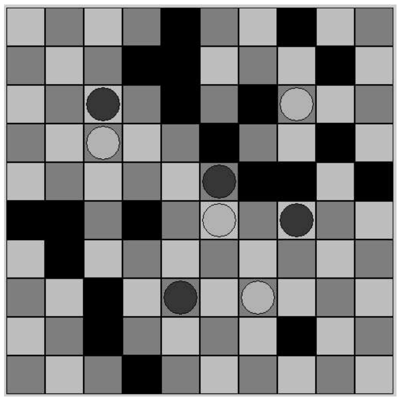
Fig. 2. An Amazons mid-game position - darkened squares are blocked squares

The main goal when playing the game is both to reduce the mobility of the opponent's Amazons, and to create territories (see figure 2). To this extent, Amazons is a territory game, like the game of Go. A second aspect on which the game is similar with Go is its high complexity: the game is shorter

J. Kloetzer and H. Iida are with the Research Unit for Computers and Games, JAIST, 1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan, j.kloetzer,iida@jaist.ac.jp

B. Bouzy is with the CRIP5, Université René Descartes, 45 rue des Saints-Pères, 75270 Paris Cedex 06 France, bouzy@math-info.univ-paris5.fr

(maximum 92 plies, usually 60-70), but its branching factor is very high: there are more than 2000 different moves for the first player. But, unlike the game of Go, some very strong game-programs already exist, mostly because it is easier to design an evaluation function [3].

B. Solvers

Solving game problems has always been a significant part of the more general goal which is creating good game-playing programs. This concerns both endgames positions and local problems and, in a more general class, game-solving. Computer programs always have an upper hand at that task over human players because of the mathematical precision needed for this task; but it nonetheless requires good algorithms, which can be easily sorted into two classes:

- On the one hand, traditional solving methods, such as PNS and DFPN [4] used for the solving of AND/OR trees. These will be later referred as "solving methods".
- On the other hand, game playing algorithms, such as iterative deepening with an evaluation function and Alpha/Beta, which are not specifically developed to solve game problems, but can nevertheless be used for that task. These will be referred later as "game-playing methods".

In the next sections, we will present four of those solvers (two solving methods and two game-playing methods) used in the context of this article.

1) Depth-First Proof Number Search: Among these algorithms, one of the most well-known and efficient is Proof-Number Search, developed in 1994 [5] (PNS). The main idea of PNS is to make use of Conspiracy Numbers [6], in the present case proof numbers (pn) and disproof numbers (dn), to search first nodes whose value is easier to prove or disprove. The search in PNS is conducted until the value of the root (its pn and dn) is found by iteratively expanding a most proving node, *i.e.* one of the easiest node to prove or disprove.

DFPN is a fork of PNS developed in 2002 [7] using the same theory and algorithm. It notably adds two threshold numbers for each node during the exploration of the tree to facilitate the search of a most proving node. DFPN is known to have solved some of the longest Shogi problem created (like "Micro Cosmos" [7]), thus accrediting its performances. The algorithm, being depth-first, is also less memory-intensive. [4]

2) Weak Proof Number Search: WPNS is another variant of PNS recently developed [8] whose performances have been shown slightly better than those of PNS. It has the main advantage of better handling the presence of Directed Acyclic Graphs (DAG) in the explored tree. The main difference with the original algorithm is the replacement of the following formula used to calculate proof numbers for an AND node N:

$$pn(N) = \sum_{c=childNode(N)} pn(c)$$

with this one:

$$pn(N) = \max_{c=childNode(N)} pn(c) + |N.unsolvedChilds| - 1$$

Also, the formula calculating disproof numbers for OR nodes is replaced in the same way.

The two variations of DFPN and WPNS being independent of each-other, it is obviously possible to adapt both these methods to create a Depth-First version of WPNS.

3) The Alpha/Beta method: Despite not being specifically designed for solvings, game-playing engines based on iterative deepening, an evaluation function and the Alpha/Beta method [9] can also be used for that task. The latter's main purpose is to cut parts of huge game-trees and it is nowadays universally used in Tree-Search based game playing engines.

4) The Monte-Carlo method: The basic idea of this last method in game programming is to evaluate positions by an approximation of, usually, their average percentage of win. This is achieved by running a huge number of random games from a given position and by averaging their results, thus computing the approximation we need. It has been proposed in 1990 by Bruce Abramson [10].

The Monte-Carlo method has been used in deterministic games since already 1993 [11], but did not have very good results for this class of games until 2006, with the creation of the UCT algorithm [12]. The latter is used to improve the performances of a Monte-Carlo engine: instead of playing completely random games, it performs a form of tree-search near the root of the tree to focus on more promising nodes, which in turns helps to find the best move from a position.

As for the Alpha/Beta method, despite not being designed for solving, the Monte-Carlo method can be used to find solutions to game problems. Its main weakness is obviously that it will return an approximation of an evaluation, while when solving problems we deal with exact solutions, resulting in an inability to "catch" the mathematical precision beneath that task. But, despite this drawback, this method has already shown some good results for problem solving [2].

III. Methodology

When considering Amazons-problem solving, two difficulties spring immediately to mind. At first, being a young and less played game, the game of Amazons does not have much theory to support it. More importantly, there is not yet any good reference or book of problems, like there are Chess problems, Tsumego or Tsumeshogi books. In addition, since it takes a great deal of knowledge and time to build "good" problems with only one solution, we will most likely cope with problems allowing multiple solutions.

Second, the game of Amazons usually reach a point where the board is separated into distinct regions, each one containing Amazons which can no longer interact with other Amazons in other regions: we will call these regions sub-games (see Figure 4: the upper board contains 3 sub-games, with respectively 1, 3 and 4 Amazons). Knowing the value of each of those sub-games and using Conway's theory of partizan games [13], we can then in turn know the value of the main game and which sub-game should be played first. But where the goal of the game is just two-valued (win/lose), the goal of each of these sub-games is to make the maximum number of points of them, which means that their goal is multi-valued. As such, the problems we will cope with will also be multi-valued. This raises two issues:

- Traditional solving methods such as PNS usually not used with multiple-valued trees, only two-valued.
- Because they are not designed as solving methods, algorithms such as Alpha/Beta or Monte-Carlo usually cannot give the theoretical value of a position.

In the next two sections, we will see first how to build a correct database of problems for the game of Amazons, and then consider how we can assess and compare different solving algorithms in the most fair way possible, having the last two issues in mind.

A. Building a problem database

Unlike more popular games such as Chess, Checkers, Shogi or Go and despite some previous work on Amazons endgames [14], there is not any easily accessible database of problems for the game of Amazons. Moreover, such collections of problems are usually written by human experts of the game and provide interesting problems with a good quality. Such problems usually have a unique solution (or few of them). That aspect is important in the field of game-programming because it allows us to check easily if a program's answer is good or not.

For the purpose of this study, we had to build such a problem database. But we need to define first what kind of problems we will deal with.

1) Amazons problems: The main part of the game which can provide us with interesting problems for the game of Amazons is the endgame. The main difference with other traditional game is that the goal in Amazons endgames is to fill (if only one side is involved) or to gain (if both sides are involved) territory: solving a problem consists in finding out the two values of the position, for each player moving first and, in a game-oriented perspective, one of the best moves.

To build this problem database (which is, in fact, a positions database), we focused on some distinct aspects:

- To help analyzing the strengths or weaknesses of every assessed method, problems should be of various sizes and contents.
- Difficulty of problems (mostly in term of size) should be capped. Not because too difficult problems would be uninteresting, but because we are dealing with limited time and with more practical than theoretical situations, and also because we need to solve the problems before utilizing them in experiments.

2) Creating the problems: Since creating a problem database by hand was out of the question, there were basically two main possible choices to create the problems:

- Use a routine to create positions automatically
- Extract positions from Amazons game records

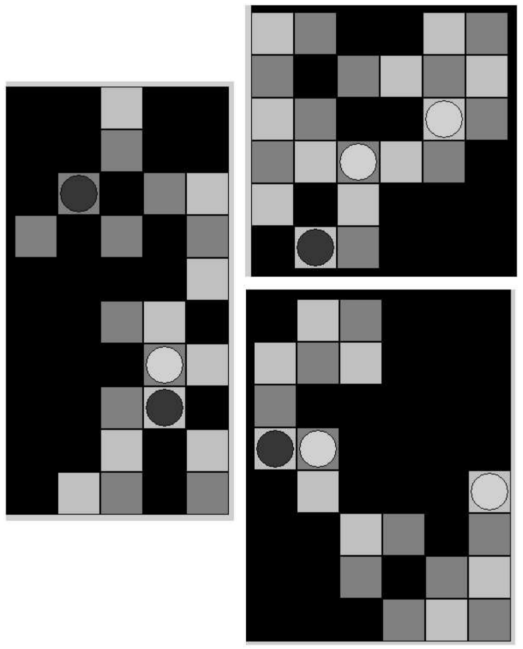

Fig. 3. Examples of Amazons end-games positions

Since there is no easy way to create a "position-making" routine and that we wanted to cope with practical situations, the second solution was chosen. The procedure used was the following:

- Fix a set of features for the problems to extract (number of Amazons, number of empty squares, possibility of movement of each player...)
- Read game records until a territory containing the correct set of features is created (see Figure 4)
- Extract the position corresponding to the territory with the position of the Amazons, and go back to reading

Each position extracted can give birth to two problems, considering which players plays first. This simple procedure, provided that we have a various set of game records and a correct set of features, should allow us to extract various non-artificial positions and to attain our goal of creating a good problems database.

B. Assessing the solvers

If we want to compare fairly two different class of solvers, solving methods and game-playing methods, for the game of Amazons, some difficulties arise. Among them, the fact that PNS-based methods are suited to solve two-valued problems while Amazons problem is multi-valued will be discussed in the first sub-section. We will then see how to assess correctly game-playing methods in the second subsection, and finally how to compare these two classes of solvers in the third one.

1) Handling multi-valued trees with PNS and PNS-based methods: PNS is usually used for two-valued trees: Yes - Win, and No - Lose. However, as presented by Allis in his paper about PNS, it does not mean that the algorithm cannot handle multi-valued trees. It just takes more time. Basically, if we want to know if a game can be won by at least N points,

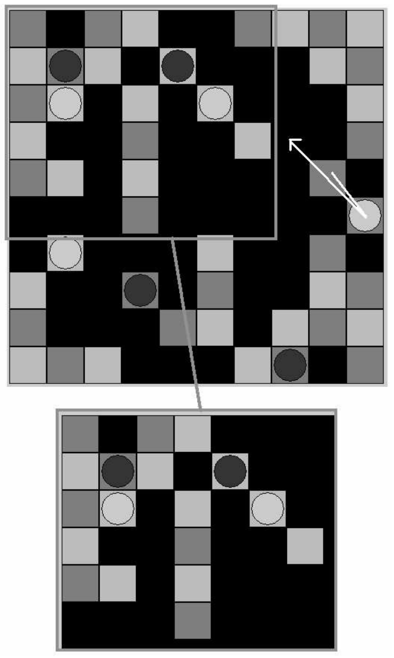

Fig. 4. In search for positions containing 2 white and 2 black Amazons. After the last move (showed in white), the part on the top left is isolated from the rest of the board and contains the correct number of Amazons for each player: it is extracted and saved in the database.

```
function getValue(position)
  Result initialResult = (Win by 1st player, 0
      point);
  int modifier = +1;
  if ( not(solve(position, initialResult)) )
    initialResult = (Win by 2nd player, maximum
        Theoretical Score of (position));
    modifier = −1;
  end if;
  Result finalResult;
  do
    finalResult = initialResult;
    initialResult.score += modifier;
  while ( solve(position, initialResult) == true );
  return finalResult;
end function

function solve(position, result)
//Consider final nodes with value strictly inferior
     to result as no, others as yes
//Perform a traditional two−valued search and
     returns the value of the root (true or false)
     or timeOut
end function
```

Fig. 5. Pseudo-code for multi-valued 2-players game solving

we should consider all final nodes with values strictly inferior to this result as NO nodes, and all others as YES nodes. On the other hand, if a position cannot be won, we want to know by how many points we would lose it, which value we can get by using the same idea. This leads us to the algorithm given in Figure 5.

Still, this procedure takes much more time than the only solving of a two-valued game-tree to find out if it is a win or a loss for the first player. If we give the same quantity of time to PNS-based solving methods and to game-playing methods, the ensuing comparison will not be fair to solving methods. So, instead of fixing a limit of time for the whole procedure, we should give it the same quantity of time for each iteration of the loop. This way, we will assess solving methods by the maximum score for which they can solve the position given a certain quantity of time.

Also, we will normally need to explore the game tree until both players have passed: if one player (P) pass, his opponent has to play until the end to determine the score of the game, while player (P) pass until the end. However, this is not always necessary, since it is possible to stop the search sometimes as soon as one player has passed. If we want to know if a position can be won by the First Player (FP) with a minimum result of (Win of player P, score N), we can stop searching at the following moments:

- If player P just passed, the search is over: the expected result cannot be proved. If P was the first player to move (that is, if P = FP), then the current node is a NO node. Otherwise, it is a YES node.
- If the opponent of player P just passed for the Nth time, the search is over: the result is proved. In this case, if P = FP, then the current node is a YES node, and otherwise a NO node.

Because they are based on the same model and have the same limitations as PNS, the procedure and improvement presented here can also be used to assess methods such as DFPN or WPNS.

2) Assessment of game-playing methods: Traditional solving methods such as PNS are able to return the final value of the root. Other algorithms such as Alpha/Beta are game playing methods: they will return an evaluation and/or a move, but we are usually not sure about how this evaluation is related to the score; this means that the only exploitable data we have is the move returned by the algorithms. To be able to compare game-playing methods between each-other and with various solving methods, we need to know how good the move given is compared to an optimal solution.

If we have an information telling us that the move given by such methods is optimal, then the search is over. But it can happen that this information is missing. In this case, we should use a judge program to compute the exact value of the position resulting from playing the move on the initial position, and see how this new value compare to the value of the initial position. It obviously cannot be better: either they are equal, in which case we know that the method assessed gave a correct solution; or they are not, in which case we also know the difference, in terms of a score difference, between the solution given by the assessed method and an optimal solution.

3) Making a fair comparison: Following these two procedures, we can get assessments both for solving methods and for game-playing methods. However, the assessments resulting from these procedures are not based on the same source of information: the assessment of a move given by a method such as PNS is given by the method itself, whereas the assessments of a move given by a game-playing method is given by a judge program. Thus it cannot really be considered as fair.

Moreover, when assessing a solving method such as PNS with limited time, it can happen that it is able to evaluate that a certain move is sufficient to give a certain score, but is unable to show that it is in fact the optimal move and leads to a better score (see Figure 6 for an example of such a situation).

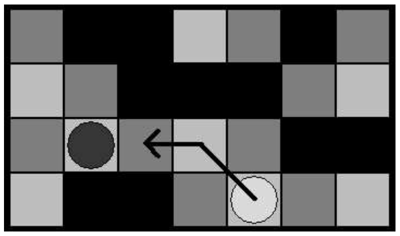

Fig. 6. The move shown can win the position, but it is easier to prove this than the fact that it also maximizes the score (5 in this position)

To solve this difficulty, the easiest solution is to treat solving methods the same way as game-playing ones: that is, instead of using the evaluation given by the solving method itself, to use a judge program to assess each method instead of using it only for game-playing methods.

IV. Results and discussion

A. Experiments settings

All experiments presented here were made on Pentium 4 3Ghz equipped with 1Gb of RAM memory.

1) Solvers: Four solving and game-playing methods were considered into these experiments:

- A DFPN solver, based on the description given in Nagai's thesis [7], with the improvement presented in section III-B.1.
- A depth-first version of a WPNS solver [8], based on the previous one
- An iterative deepening Alpha/Beta engine, using an evaluation function based on the accessibility [3]
- A Monte-Carlo engine with UCT, based on our program Campya [15]

The program used to get the correct values of positions (the judge program) is the same DFPN solver as the one used in the experimentations, except that it does not have the limited time used for the comparison.

2) Problems creation: The game records used for the problem extraction were obtained by self-play games of our program, Campya. The time for the games was set to 5 minutes for each side, so that we would have games both of a good level and various enough.

The extraction of positions from these game records was based on the number of Amazons present in those positions: We focused on extracting positions containing at most 4 Amazons, with at least one for each player.

Problems too simple were also removed from the database. To perform this task, we relied on three characteristics of the problems:

- Its size (in terms of number of empty squares)
- The mobility of each player, that is the number of moves playable by each player from depth 0
- The possibility of blocking the second player in just one move

We decided to cut arbitrarily problems whose size was inferior to 10, as well as those where one player had less than 5 possible moves, and finally problems solved by blocking the second player in one move. This let us with around 900 problems that we were able to solve in reasonable time to get their exact value with our judge program, with maximum depth (number of empty squares) varying from 10 to 24 (see Figure 7). Examples of those problems are given in Figure 3.

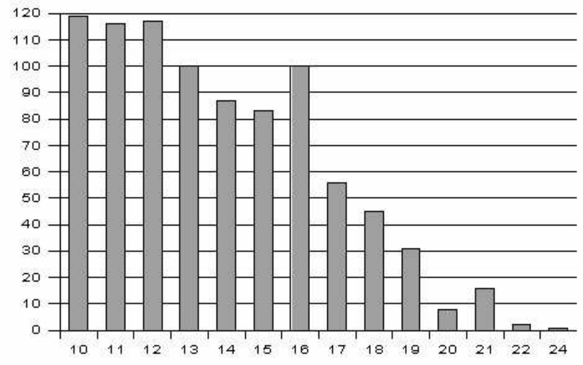

Fig. 7. Number of problems generated per number of empty squares

3) Procedure: The 4 solvers presented above were ran on our problem database using the following procedure. For each problem (P) of the database:

- Get the theoretical value (V) of the problem using the judge program
- For each method to be assessed:
  - If it is a game-playing method, give it a time limit of 10 seconds to chose a move
  - If it is a solving method, give it a time limit of 10 seconds for each iteration of its loop (See section III-B.1) and return the best move computed
  - Get a move from the assessed method
  - Use the judge program to get the theoretical value of the position obtained from playing the given move in the initial position, and compare it to V

B. Experiments and Results

Results of the comparison are given in Table I and are quite clear: the Alpha/Beta method outperforms all its competitors, while the most recent version of PNS tested, WPNS, gives better results than DFPN.

However, these percentage of solved problem do not summarize everything. The Figure 8 gives us the percentage of problems solved by each method considering the size of the problems: it appears that, although the number of

TABLE I

PERCENTAGE OF PROBLEMS SOLVED FOR EACH ASSESSED METHOD

Method	Percentage
DFPN	81.84
WPNS	87.74
Monte-Carlo	87.51
Alpha/Beta	98.52

TABLE II

STATISTICS ON UNSOLVED PROBLEMS

Method	Percentage unsolved	Percentage badly solved	Average scoring error
DFPN	9.49	6.74	4.97
WPNS	6.16	5.59	5.43
Monte-Carlo	0	12.49	1.67
Alpha/Beta	0	1.48	1.15

problems solved by both WPNS and Monte-Carlo are similar, the solving abilities of Monte-Carlo do not vary that much with the depth, on the contrary of those of WPNS and DFPN (the peak at the end cannot be really considered because there is only one problem of size 24).

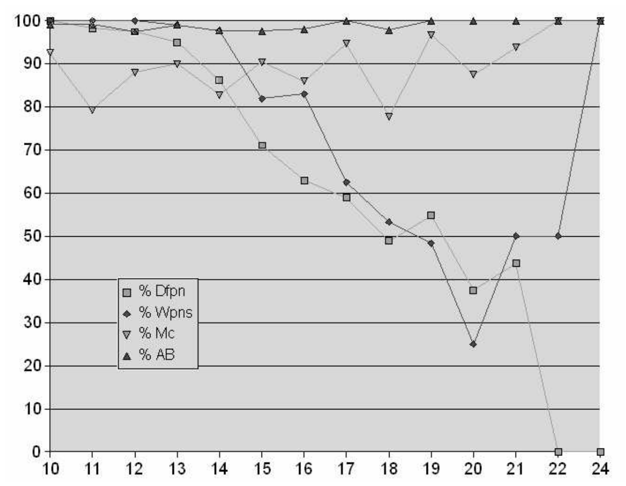

Fig. 8. percentage of problems solved for each method per depth

Finally, we should also consider the average error made by each method compared to the correct solution. Parts of the problems have been totally unsolved by both DFPN and WPNS, because of insufficient time, in which case no solution at all is given. But for other problems, as well as for Monte-Carlo and Alpha/Beta, an incorrect solution is sometimes given: in this case, it is possible to compute the number of points of error compared to an optimal solution. This value is given in table II. It appears at first clearly that the dominance of the Alpha/Beta method is unquestionable, but also that, even if it was not able to solve all the problems, the mean error made by the Monte-Carlo method is quite low. This shows that, despite its inability to catch very precise or mathematical values such as an exact score, the performances of this method are still very good for the task of solving Amazons problems.

C. Results discussion

From these results, it is cleare that the performances of PNS-based methods (DFPN and WPNS) are quite disappointing. Although DFPN is able to solve with a high accuracy short problems, its performances drop considerably when the depth of the problem goes over 13 or 14. It seems that this factor, combined with the high branching factor of Amazons, is the main reason for its lack of performances at high depth.

Since the game of Amazons permits the apparition of transpositions, and so of Directed Acyclic Graphs into the game-tree, we had expected WPNS to perform better than DFPN on this test-bed, and were not disappointed: although the results of WPNS are not as good as those of Alpha/-Beta, the improvement in performances over DFPN is quite clear. But, when comparing directly the results of DFPN and WPNS, it appears that DFPN was able to get strictly better results than WPNS on 17 problems, while WPNS got strictly better results on 57 problems: This shows us that WPNS is not strictly better than DFPN, and that the method should be tested more deeply before saying that it is a clear improvement over PNS.

Game-playing methods (namely Alpha/Beta and Monte-Carlo), on the other hand, got much better results, especially Alpha/Beta. The mistakes made by Monte-Carlo should obviously not be forgotten, but all in all, it also proved pretty strong at solving Amazons problems, especially considering that it is not designed at all to solve problems. These results tend to confirm those of another recent study [2], which showed good results for the Monte-Carlo method to solve Go problems. However, their results for the Alpha/Beta method, outperformed both by DFPN and badly by Monte-Carlo, do not concur with the present ones.

The most plausible explanation to this behaviour would be the presence, in the case of our test-bed, of a good evaluation function in the Alpha/Beta engine. Although it is possible to create a good evaluation function for the game of Amazons, even if it is simple, this task is much more complicated for the game of Go, even for a simple task such as endgames. So even if both algorithms (Iterative Deepening with Alpha/Beta and Monte-Carlo with UCT) are powerful enough by themselves, it clearly seems that using an evaluation function is a necessary need for the game of Amazons, if we want to build a strong game-playing engine (which also confirms results presented in [15]).

V. CONCLUSION AND FUTURE WORKS

We have built for this article an Amazons problems database, and presented the results of our comparative study for different solvers applied to Amazons endgames. Unexpectedly, traditional solving methods or their improvement performed less good than game-playing methods. We attribute these results mainly to the presence of an evaluation function in the latter, which helps focus the search in the correct direction. This work also shows the importance of such a function for the game of Amazons.

It remains now to show until which complexity (possibly qualified by the depth) of problems these game-playing methods are efficient, the test-bed providing sufficient problems only up to depth 20. Also, we would like to improve our Monte-Carlo engine so that it would be able to catch the mathematical precision present behind these problems and not make mistakes which, even if they are of minor importance in term of score, can decide of a win or a loss at high level of play.

Appendix

The problem database built for this study can be found in XML format at the following adress: `http://www.jaist.ac.jp/~s0720006/AmazonsProblemsDatabase.txt`

References

[1] "Computer beats Pro at U.S. Go Congress," `http://www.usgo.org/index.php?%23_id=4602`

[2] P. Zhang, and K. S. Chen "Monte-Carlo Go Tactic Search," in *Information Sciences 2007*, 2007, pp. 662–670.

[3] J. Lieberum, "An Evaluation Function for the Game of Amazons," *Theoretical computer science 349*, pp. 230–244, 2005.

[4] M. Sakuta, and H. Iida, "Advances of AND/OR Tree-Search Algorithms in Shogi Mating Search," *ICGA Journal vol. 24-4*, pp. 231–235, 2001.

[5] L. V. Allis, et al. "Proof-Number Search," *Artificial Intelligence, vol. 66-1*, pp. 91–124, 1994.

[6] D. A. McAllester, "Conspiracy numbers for min-max search," *Artificial Intelligence 35*, pp. 287–310, 1988.

[7] A. Nagai, "Df-pn Algorithm for Searching AND/OR Trees and Its Applications," *Ph.D. Thesis*, Tokyo University, 2001.

[8] T. Ueda, T. Hashimoto, J. Hashimoto, and H. Iida, "Weak Proof-Number Search," in *Proceedings of the Conference on Computers and Games 2008*, To be published.

[9] D. E. Knuth, and R. W. Moore, "An Analysis of Alpha/Beta Pruning," *Artificial Intelligence Vol. 6, No. 4*, pp. 293-326, 1975.

[10] B. Abramson, "Expected-outcome: a general model of static evaluation," *IEEE transactions on pattern analysis and machine intelligence 12:22*, pp. 182–193, 1990.

[11] B. Brugmann, "Monte Carlo Go," *Technical report*, Physics Department, Syracuse University, 1993.

[12] L. Kocsis, and C. Szepesvari, "Bandit based monte-carlo planning," in *Proceedings of the 15th International Conference on Machine Learning (ICML)*, 2006, pp. 282-293.

[13] J. H. Conway, *On Numbers And Games*. Academic Press, 1976.

[14] M. Muller, and T. Tegos, "Experiments in Computer Amazons," *More Games of No Chance*, Cambridge University Press, 2001, pp. 243–260.

[15] J. Kloetzer, H. Iida, and B. Bouzy, "The Monte-Carlo approach in Amazons," in *Proceedings of the Computer Games Workshop 2007*, 2007, pp. 185–192.