Playing Amazons Endgames

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

ABSTRACT

The game of the Amazons is a fairly young member of the class of territory-games. Since there is very few human play, it is difficult to estimate the level of current programs. However, it is believed that humans could play much stronger than todays programs, given enough training and incentives. With the more general goal of improving the playing level of Amazons programs in mind, we focused here on the playing of endgames situations. Our comparative study of two solvers, DFPN and WPNS, and three game-playing algorithms, Minimax with Alpha/Beta, Monte-Carlo Tree-Search and Temperature Discovery Search, shows that even though their computing process is quite expensive, traditional PNS-based solvers are best suited for the task of finding moves in a subgame, while no specific improvement is needed to classical game-playing engines to play well combinations of subgames. Even the new Amazons standard of Monte-Carlo Tree-Search, despite showing often weaknesses in precise tasks like solving, handles Amazons endgames pretty well.

1. INTRODUCTION

The game of the Amazons is a young territory-based game with a game tree size[3] comprised between those of Chess and Shogi (Japanese Chess). Like for Shogi, several strong programs already exist but cannot yet consistently beat strong human players.

Like the game of Go in the category of territory games, the game of the Amazons has witnessed the arrival of Monte-Carlo Tree-Search (MCTS) in the field of game programming, and new strong programs have already emerged (Kloetzer, Iida, and Bouzy, 2007; Lorentz, 2008), mainly thanks to the huge quantity of work that was done to improve MCTS for the game of Go. But these studies often focus on making a strong game-program able to play well during a whole game; few worked on improving some specific parts of the game like the endgame.

For the purpose of building a strong Amazons program, playing well in the opening and in the middle game are not sufficient. Playing optimally in the endgame is also very important if one is to consider beating top human players. However, among techniques varying from precise search engines such as DFPN to sample-based such as MCTS, it is not clear which is optimal to tackle the problem of the endgame in the game of the Amazons. MCTS has been shown effective for solving Go problems (Zhang and Chen, 2007) but this method is also nowadays the standard for the game itself.

In this paper, we will study traditional techniques, DFPN (Nagai, 2002) and minimax with Alpha/Beta (Knuth and Moore, 1975) as well as newer ones such as MCTS (Coulom, 2007; Chaslot *et al.*, 2008) with UCT (Kocsis and Szepesvari, 2006), WPNS (Ueda *et al.*, 2008), and Temperature Discovery Search (TDS) (Müller, Enzenberger, and Schaeffer, 2004), with the goal of finding the one leading to the best playing of both single subgame situations and combinations of subgames. The main goal still being to build a strong game-playing engine for tournament conditions by adding to it a powerful endgame-playing engine, we focus here on realistic endgames situations, not specifically balanced for one player, and playing in limited time. We shall also see if the new Amazons standard of MCTS (Kloetzer *et al.*, 2007; Lorentz, 2008) can keep up even with this difficult task.

We will introduce in Section 2 basics about Amazons subgames and how to play them well or solve them. Sec-

[1]Research Unit for Computers and Games, JAIST, j.kloetzer,iida@jaist.ac.jp

[2]LIPADE - UFR de mathématiques et d'informatique, Université René Descartes, bruno.bouzy@parisdescartes.fr

[3]Although no clear estimation has been made, the game tree complexity of the game of the Amazons can be estimated to around 10^{165} (Avetisyan and Lorentz, 2003)

tion 3 will present our experiments to tackle the task of playing well in a single subgame, while the experiments of Section 4 will deal with the task of playing well in a combination of subgames. The conclusion follows in Section 5.

2. AMAZONS SUBGAMES

2.1 The game of the Amazons

The game of the Amazons, also called Amazons, is a two-player deterministic game with perfect information. It is usually 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 Amazon, in any direction, any number of squares away (see Figure 1 for a move example). From this point onward, the square on which the arrow lands is blocked, which prevents any further move or shot on or through that square. The first player unable to move loses the game, at which point it is usually agreed that the score of his opponent is the number of moves he can still make after the pass.

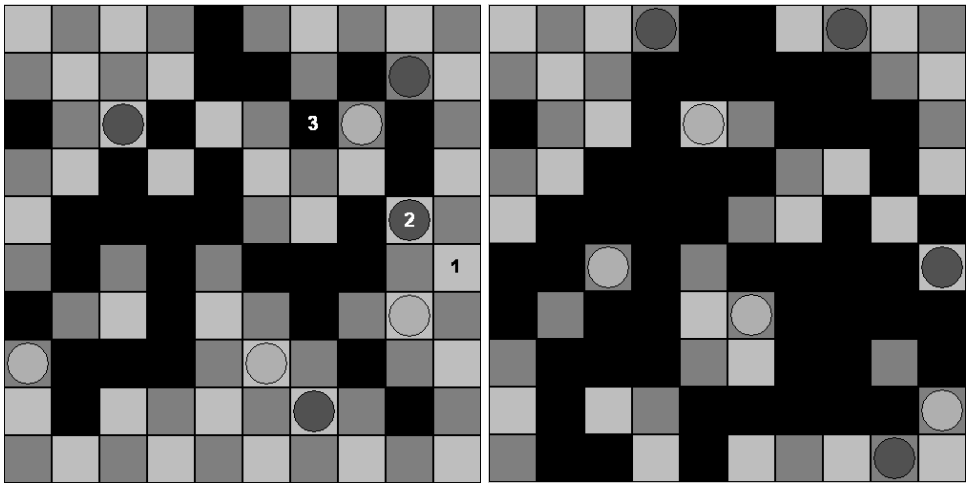

Figure 1: Left: An Amazons middle game position; the last move was 1-2 (Amazon move) + 3 (shot); Right: An Amazons endgame position

Some strategical aspects of the game include reducing the mobility of the opponent's Amazons while improving its own and creating territories (the position on the right of Figure 1 has two territories on the left, one for each player). 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 (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 strong game-programs already exist, mostly because it is easier to design an evaluation function (Hashimoto *et al.*, 2001; Lieberum, 2005).

The game of the Amazons usually reaches a point where the board is split into distinct regions, each one containing Amazons which can no longer interact with other Amazons in other regions: we will call these regions subgames. For example, the position on the right of Figure 1 has four subgames: two one-player subgames on the left side, one subgame on the bottom right, and the last one in the center and around. To play well in such a position, it is necessary both to be able to play optimally (to get the best score) in each subgame and to be able to choose the best subgame to move into. Since those have no interaction whatsoever with each other, it is possible to apply combinatorial game theory (Conway, 1976) to know the value of the main game and which subgame should be played first.

2.2 Solving subgames

Solving an Amazons subgame falls into one of two categories, depending if the subgame contains one or more Amazons for only one player or for both. With only one player, the goal is to fill the territory of the subgame with the maximum possible number of moves. The difficulty then appears in the case of defective territories, in which the configuration of the Amazon(s) and the arrows does not allow the player to fill every square of it. Defective territories have been investigated in detail in Müller and Tegos (2002). Now, if both players have Amazons in the subgame, and if this subgame is considered in isolation from other parts of the game, the goal is to get the maximum moves out of it compared to the number of moves that the opponent will get. Quite often, the situation ends with two or more one-player subgames, and the player with the biggest territory will then win.

The goal of two-player subgames being multi-valued and not two-valued, traditional and/or search methods such as PNS cannot be used as-is. Instead, we must modify their search process as described in Allis, van der Meulen, and van den Herik (1994): given a result to prove (e.g. white wins by 5 points), we tag leaf nodes as true if their value is equal or superior to this result, and false otherwise. The search for the best result is then an iterative process: we try first to prove the minimum possible result (in Amazons, a loss for the first player by a number of points equal to the number of empty squares in the subgame), then increase it until we can disprove the result to try. The value of the subgame is then the latest result that the search was able to prove. This obviously requires us to be able to order and enumerate the different possible results, which is fortunately the case for all score based games similar to the game of the Amazons.

Specifically for the game of the Amazons, several improvements can be made to speed up the search. This consists mostly in tagging true or false internal nodes of the search tree, either because the result is already proved or because it will be impossible to prove it. Considering a position where we want to prove that a player P can win by N points, we propose the following rules:

- After player P passes, the node can be tagged as false.
- If player P has been able to play N moves after a pass of his opponent, the node can be tagged as true.
- If player P is to move while his opponent did not yet pass and there are strictly less than N empty - or reachable - squares on the board, the node can be tagged as false.

2.3 Playing Amazons endgames

Playing well in Amazons endgames supposes to play well for two tasks: the first one, as presented above, is to play well into a single subgame, be it one- or two-player. For this task, it is possible either to use a traditional solver (adapted with the procedure presented in Section 2.2) like Proof-Number Search (PNS) (Allis *et al.*, 1994) or some of its variants, or to use a more traditional Amazons engine, be it a traditional Alpha/Beta engine (Lieberum, 2005) or a more recent MCTS engine (Kloetzer *et al.*, 2007; Lorentz, 2008).

Next to playing well in one subgame, there is the problem of what move to select if we play in a position composed of several subgames. Two methods can be used here: the first one is to use traditional game-engines like presented before to handle the task of choosing the right subgame while at the same time choosing a move. The second one is to choose a subgame first and then use any method to select a move inside the chosen subgame. For example, the Hotstrat heuristic consists in playing into the subgame which has the highest temperature in the sense of combinatorial game theory. Simply put, a subgame is hotter when it is more urgent to play in it compared to other subgames. Precise computation of the temperature of each subgame is unpractical, but it is possible to use an algorithm such as Temperature Discovery Search (TDS) (Müller *et al.*, 2004) to get an estimation of the temperatures we need, then perform a local - and thus deeper - search inside the subgame with the highest temperature to find the best move in it. Since they go together we will often mix the heuristic with the implementation in the next sections, thus calling TDS the combination of Hotstrat, TDS and a local search.

3. PLAYING RIGHT IN ONE SUBGAME

The following sections will describe our experiments to tackle the problem of playing well into a single subgame.

3.1 Experiment settings

Amazons is still a young game for which no extensive knowledge database or book exists. More specifically, there exists no database of problems suited to our task. For the purpose of this experiment, we created a database of positions extracted from game-records of self-play of our program CAMPYA. Reading one game-record, a position was extracted as soon as it corresponded to the criterion previously defined (such as: "One Amazon against two").

Following this procedure, we created a database of around 1300 realistic positions. These positions are of size (number of empty squares) 2 to 50, and each of them gave birth to 2 problems, with each side going first. Each problem was solved using a simple PNS solver given unlimited time. We had, however, to reject around half of the problems either because they were in insufficient number to represent their size or mostly because we were unable to solve them in reasonable time. This left us with approximately 1300 remaining problems (not to be confused with the original 1300 positions) of size 2 to 21. The distribution of those problems can be seen in Figure 2.

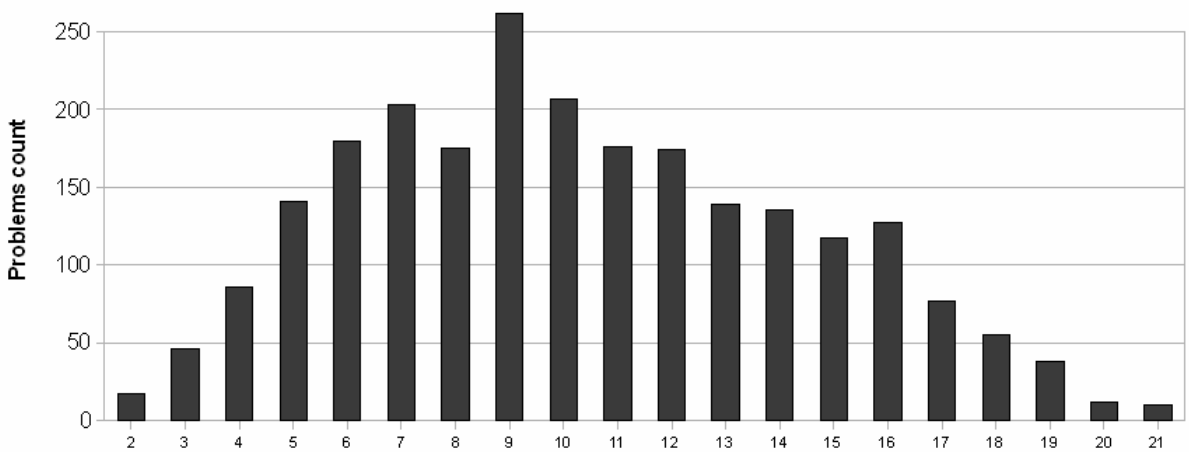

Figure 2: Problems count by size

For this experiment, we compared 2 traditional solvers and 2 game-playing engines. The first two are a traditional DFPN solver (Nagai, 2002) and a depth-first version of a WPNS solver (Ueda *et al.*, 2008), a recent work on PNS showing promising results. Both are using the Amazons specific improvements presented in Section 2.2. The other two are a traditional Alpha/Beta game-playing engine (using Iterative Deepening, a Transposition Table, move ordering mostly based on the moves evaluations, and forward pruning such as described in Avetisyan and Lorentz (2003)), and an MCTS engine based on our program CAMPYA (which main features are described in Kloetzer *et al.* (2007)). Both use the same evaluation function based on queen-distance as an approximation of the territory such as described in Lieberum (2005).

3.2 Assessing the different methods involved

All four algorithms were evaluated on the database of problems presented in Section 3.1. Each of the algorithms was given the same quantity of time: 5, 10 or 20 seconds depending on the setup, to find a move for each problem.

Each of the algorithms was assessed the same way: for a given problem, we first get a move out of each of them (the best move chosen by a game-playing engine, or the move proving the best result obtained in the iterative process for the solvers). Then, if that move is not found in our database, we evaluate the position resulting from playing it with a more powerful solver - a traditional PNS solver given unlimited time - and compare it to the value of the original problem. The value can only be equal - in which case one of the best moves was chosen - or less - in which case we can set the difference between both results as the error made by the algorithm assessed.

Using the procedure presented in Section 2.2 to solve a problem we get two informations: the best move selected by the solver and the last result that the solver was able to prove in its iterative process. This is, in a way, the evaluation of the problem by the solver. This evaluation can be exact or just a minimum. We could assess both our

solvers - DFPN and WPNS - using this evaluation; however, we decided against it and chose to evaluate the same way solvers and game-playing engines as presented above. The first reason is to keep the comparison clear by comparing similar informations, and the second is because there are situations in which a solver can easily prove that a move can win by a certain score but not prove that this move is optimal for the best score (see Figure 3 for an example). In this case, assessing a solver by its returned evaluation would be unfair to it. Since our first goal is to play well in the endgame, it made sense to only consider the move returned by a solver to assess it.

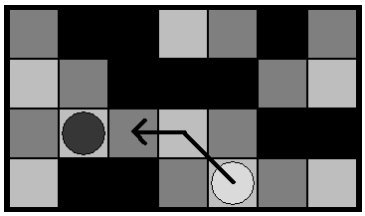

Figure 3: Proving that this move wins in this position is less difficult than to prove that it indeed leads to a score of 5 points

3.3 Results

Table 1: Percentage of problems correctly answered for each algorithm and (average error) for those wrongly answered

	5 seconds	10 seconds	20 seconds
DFPN	92.09 (3.49)	93.65 (3.26)	95.12 (2.97)
WPNS	95.92 (2.39)	97.18 (2.34)	98.28 (2.37)
Alpha/Beta	97.31 (1.25)	97.64 (1.27)	97.81 (1.25)
MCTS	94.74 (1.66)	94.87 (1.66)	95.41 (1.64)

Table 1 gives the percentage of problems correctly answered by each algorithm, given different quantities of time. Unsurprisingly, most of the problems being of reasonable size, they are easily tackled by all of the algorithms.

Compared to the standard which is Alpha/Beta search, the performances of MCTS are somehow disappointing: it makes more errors and those are usually more important in terms of the score difference to the optimal value. It seems that MCTS scales slightly better, but 20 seconds for a move is already close to what a program will get in the endgame in tournament conditions.

However, considering that the iterative procedure used to solve a multi-valued problem is quite time-consuming, the performances of both PNS-based solver are very good. Traditional Proof-Number Search (here represented by DFPN) shows some weaknesses but shines with the improvements given by WPNS. In practice, we observed that most of the iterations take little if not no time at all and that both solvers usually use most of their time on the most difficult iteration. Also, the approximation made by playing the move given by the last iteration solved seems to function pretty well as long as the solver gets enough time. Otherwise, it may make some pretty bad mistakes, as shown by the average error (in brackets in Table 1). Finally, the improvements to the solving process presented in Section 2.2 were necessary: the performances of these solvers dropped by more than 10% if for instance the last of the three was missing.

Now, looking at the more detailed results of Figure 4, we can also see that the performances of the PNS-based solvers tend to slowly decrease after a certain size (10-11) while still being better up to around size 15. On the other hand, the performances of the game-playing engines, even if weaker for even smaller problems and always irregular, tend to stay stable whatever the size and are better for size 16 and onward. Also, the simple fact that we had to reject around half of our problems of size 20 and more because we were not able to solve them in reasonable time (problems not included in the present results, mentioned in Section 3.1) confirms the lack of performances of PNS based algorithms to handle bigger size problems.

There are unfortunately few problems of size 18 or more in our database, so the results for these sizes are far from being statistically representatives. The fact that all problems of size 20 are perfectly solved by every algorithm

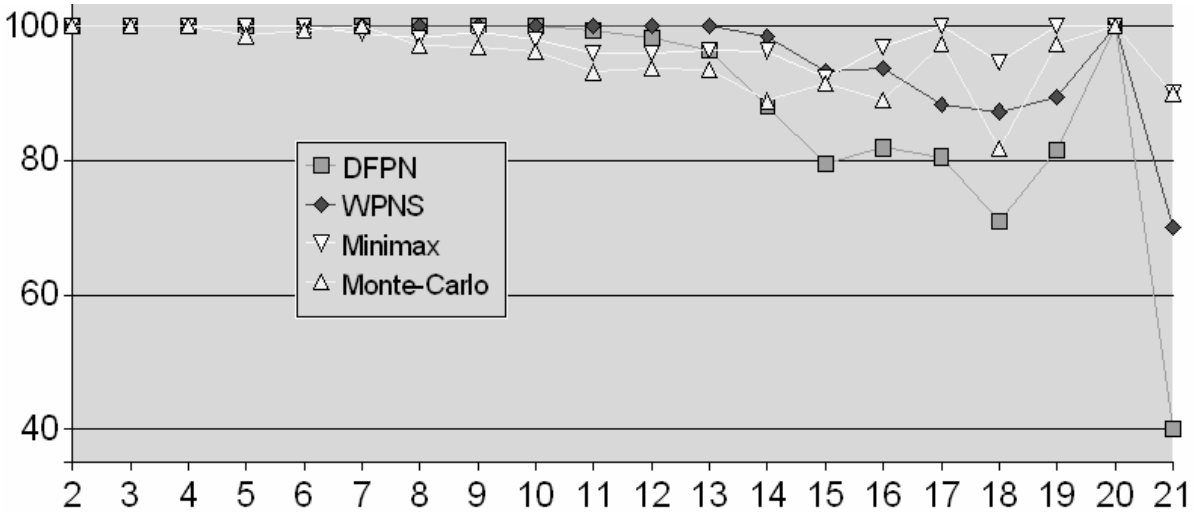

Figure 4: Percentage of problems correctly answered (20 seconds time)

for example is a fortuitous consequence from the simple fact that our 13 problems of size 20 were even easier to solve than some of the smaller ones. However, the decrease in performances of DFPN and WPNS for problems or size 12 and more is quite clear, so we will for now extrapolate for bigger sizes.

Finally, with slightly less good performances overall for all algorithms, the same conclusions can be drawn for time settings of 5 and 10 seconds. For this reason, we chose to present only the results for 20 seconds.

These results suggest that the inclusion of a solver could be a welcome improvement in any Amazons program to correctly play in the endgame once the play reaches difficult small positions. In this case, some knowledge could perhaps be added from the game-playing engines themselves, like what is done in DFPN+ in Nagai (2002).

4. PLAYING THE BIG PICTURE

Since playing the real Amazons game involves playing a combination of subgames in the endgame, we also experimented with those. The settings and results of such experiments are presented in the next sections.

4.1 Experiment settings

To create a set of positions to use for this experiment, we used combinations of positions taken from the database presented in Section 3.1. For that, we randomly associated together positions of any size to fit into a classical 10×10 Amazons board, including those of large size not mentionned in the results of Section 3 because we were unable to solve them in limited time and possible one-player subgames. We created this way 300 Amazons endgames positions, split into sets of a hundred each consisting respectively of 2, 3 and 4 subgames. The number of Amazons in each positions is not fixed and does not go over 4 Amazons for each player to respect the rules of the game.

We used three game-playing engines in this experiment: an Alpha/Beta engine, an MCTS engine (both presented in Section 3.1), and a minimax game-playing engine based on TDS and the Hotstrat strategy presented in Berlekamp, Conway, and Guy (1982). For the latter, Alpha/Beta is used after the computation of the temperatures to select a move into the hottest subgame.

We compared directly the performances of each algorithm by making them play against each other. Each pair of engines played in total 4 games for each position of our endgames set: each endgame was played with both sides playing first, each engine playing first one time. A time limit was set to 10 seconds per move for each match.

4.2 Results

The results of this experiment in terms of numbers of games won by the first player are given in Table 2 for each of the programs AB (Alpha/Beta), MC (Monte-Carlo Tree-Search) and TDS. It should be noted that the positions created for this experiment are not specifically fair and do not give the same fighting chances to both players. For this reason, we also included AB-AB matches to have a reference value to compare the other results to.

Table 2: Percentage of problems won for the first player of each pair tested

1st player	2nd player	2 subgames	3 subgames	4 subgames	All problems
AB	AB	41%	40%	51%	44%
AB	MC	40.5%	37.5%	45%	41%
MC	AB	42.5%	42%	51.5%	45%
MC	TDS	44%	42.5%	56%	47.5%
TDS	MC	40%	36%	45.5%	41.5%
AB	TDS	42.5%	41.5%	52%	45.5%
TDS	AB	39.5%	38%	48.5%	42%

From these results, we can easily infer that MCTS performs slightly better than Alpha/Beta, and that both perform better than TDS.

Unexpectedly, TDS could not get good performances against both other playing engines while it is specifically designed for this task. We associate these results to the time used in the pre-computations necessary for TDS to work correctly. The results presented in Müller *et al.* (2004) were much more in favor of TDS, but the test-bed was different: the problems used then were much more balanced and perhaps more difficult to handle with simple Alpha/Beta search, while at the same time allowing to TDS the possibility to shine.

On the other hand, the Alpha/Beta and the MCTS based engines seem to have comparable performances on most of the problems, with a slight advantage to the MCTS engine, advantage which appears bigger with more subgames. This result can be explained by the usual better performances of MCTS in more general situations and its better ability to ”catch the big picture”, and that even if it lacks the precision of Alpha/Beta or other solvers such as DFPN for playing the subgames themselves right.

Since most of the positions of the test-bed are unfair for one of the players (as can be seen by the AB/AB comparison), we cannot base our results on comparing only the percentages of victory of each engine: we also have to compare the results of each of them on each single problem. For that, given a single problem and two engines, we compared the score obtained by both engines with each of them playing first against the other and noted which got the best score. A summary of these results is given in Table 3.

This comparison tends to validate both the performances presented above for all number of subgames and, when looking at the average difference in score between each engine, the ability of Alpha/Beta based engines (both traditional minimax search and TDS) to be more precise in their search, leading to overall better average scores.

Table 3: For each pair A-B of engines: with A and B alternatively playing first, number of problems with A getting a better score than B / A and B getting the same score / B getting a better score than A. On the second line: average score difference for each of these problems.

Pair	2 subgames	3 subgames	4 subgames	All problems
AB-MC	51 / 108 / 41	44 / 103 / 53	68 / 56 / 76	163 / 267 / 170
	3,82 / - / 2,02	2,57 / - / 1,83	2,68 / - / 2,79	3,01 / - / 2,31
MC-TDS	46 / 107 / 47	60 / 107 / 33	106 / 48 / 46	212 / 262 / 126
	2,76 / - / 4,13	2,83 / - / 2,73	3,21 / - / 2,8	3 / - / 3,28
AB-TDS	49 / 114 / 37	66 / 105 / 29	109 / 54 / 37	224 / 273 / 103
	2,86 / - / 2,14	2,55 / - / 1,79	2,94 / - / 1,95	2,81 / - / 1,97

Analyzing these results through other angles, we also discovered a relation with the size of the problems for the AB-MC comparison: Alpha/Beta performs better than MCTS on bigger problems. For problems of size up to 25, MCTS performs better than Alpha/Beta on 60 problems versus 46 problems with Alpha/Beta getting better results (and 158 drawn problems), while for problems of size 30 and more, these results go to 65 for MCTS versus 76 for Alpha/Beta (and 198 drawn problems). While lots of the problems are still draws, this contrasts with the results observed before showing that MCTS performs better when the number of subgames is larger. Still, this is not contradictory, since it suggests that Alpha/Beta performs better against MCTS with 2 long subgames than with 4 shorter ones, and vice-versa for MCTS.

5. CONCLUSION

With the main overall goal of improving the playing level of Amazons programs, we have focused here on the task of playing well endgames situations. Since this problem is related to combinatorial game theory, we have compared in this study the performances of several algorithms handling two tasks: playing well single subgames or combinations of subgames. In both aspects, Alpha/Beta search with an evaluation function appears as a good standard to play correctly in Amazons endgames and does not need specific improvements to play combinations of subgames such as TDS. Nevertheless, it also appears that PNS-based solvers could be a welcome inclusion in any Amazons program to play perfectly in smaller subgames. Some progress can also be made to improve these solvers, as shown by the very good results of WPNS in this study. Finally, even if it lacks the precision of the former engines, the new MCTS engines seem better suited to play combinations of subgames, a task for which the order of play is important.

On the future of this topic, we hope to improve the precision of our MCTS engine so that it could surpass Alpha/Beta in all ways, being able to play as good in a single endgame while at the same time having better performances in playing right a whole game consisting of several subgames. On possibility could be to modify MCTS to handle the task of choosing a subgame, leaving to a more precise engine such as PNS the task to find the best move in the area in a short amount of time. Also, building a database of endgame positions could be a welcome inclusion for all of the tested algorithms to shorten the search and improve their precision.

ACKNOWLEDGEMENTS

The authors would like to thanks the four anonymous referees for all the comments and criticism they provided to improve the quality of this article.

6. REFERENCES

Allis, L., Meulen, M. van der, and Herik, H. van den (1994). Proof-number search. *Artificial Intelligence*, Vol. 66, No. 1, pp. 91–124.

Avetisyan, H. and Lorentz, R. (2003). Selective Search in an Amazons Program. *Lecture Notes in Computer Science*, pp. 123–141.

Berlekamp, E., Conway, J., and Guy, R. (1982). *Winning ways.* London: Academic Press.

Chaslot, G., Winands, M., Herik, H., Uiterwijk, J., and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree-Search. *New Mathematics and natural Computation*, Vol. 4, No. 3, pp. 343–357.

Conway, J. (1976). *On Numbers And Games.* Academic Press.

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Lecture Notes in Computer Science*, Vol. 4630, pp. 72–83.

Hashimoto, T., Kajihara, Y., Sasaki, N., Iida, H., and Yoshimura, J. (2001). An evaluation function for amazons. *Advances in Computer Games*, Vol. 9, pp. 191–202.

Kloetzer, J., Iida, H., and Bouzy, B. (2007). The Monte-Carlo Approach in Amazons. *Computer Games Workshop, Amsterdam, The Netherlands*, pp. 113–124.

Knuth, D. and Moore, R. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, Vol. 6, pp. 293–326.

Kocsis, L. and Szepesvari, C. (2006). Bandit Based Monte-Carlo Planning. *Lecture Notes in Computer Science*, Vol. 4212, p. 282.

Lieberum, J. (2005). An evaluation function for the game of amazons. *Theoretical Computer Science*, Vol. 349, No. 2, pp. 230–244.

Lorentz, R. (2008). Amazons discover Monte-Carlo. *Computers and Games, Beijing, China, September/October 2008*, pp. 13–24.

Müller, M., Enzenberger, M., and Schaeffer, J. (2004). Temperature discovery search. *Nineteenth National Conference on Artificial Intelligence (AAAI 2004)*, pp. 658–663.

Müller, M. and Tegos, T. (2002). Experiments in computer amazons. *More Games of No Chance*, pp. 243–260.

Nagai, A. (2002). *Df-pn Algorithm for Searching AND/OR Trees and Its Applications.* Ph.D. thesis, Department of Information Science, University of Tokyo.

Ueda, T., Hashimoto, T., Hashimoto, J., and Iida, H. (2008). Weak Proof-Number Search. *Computers and Games, Beijing, China, September/October 2008*, pp. 157–168.

Zhang, P. and Chen, K. S. (2007). Monte-Carlo Go Tactic Search. *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pp. 662–670.