Using Monte Carlo Techniques in the Game Ataxx

Evert Cuppen

June 22, 2007

Abstract

In this paper we will report research on an AI player for the game Ataxx. We will implement a player that uses Monte Carlo search, improved with domain knowledge and a structural enhancement. We will test this player against a player that uses the $\alpha\beta$-algorithm, which is the standard implemented algorithm for this game.

1 Introduction

In recent years, the computer has become a serious opponent for humans in many games. A very widely known example is the game of chess. In this field, the computer is able to challenge the best players in the world [4]. In other games however, the computer has never been a real threat to most human players. An example is the game Ataxx, because of the high complexity [1, 7]. It is a relatively unknown game and little research has been done so far. However, some methods have already been proposed to make the computer able to play the game [1]. These methods are mostly the minimax and alphabeta algorithm. These algorithms look for the best move from all possible moves in a depth of n moves, where n is the number of moves you want to look ahead. Also, some simple heuristics have been used to decide the computer's move.

One new approach though is an AI using Monte Carlo techniques. Monte Carlo techniques have proven to be very useful in all kinds of other domains, including games like Go [2]. It was found that such techniques are especially worthwhile when not much domain knowledge is available. The Ataxx game also belongs to that domain, so researching Monte Carlo techniques will be very interesting. This brings us to the following research question: "Can we build a strong AI player for the game Ataxx using Monte Carlo techniques?"

The remainder of the paper is structured as follows. First, we will explain the rules of the game in section 2. In this section we will also discuss some properties of the game. In section 3 we will explain the different stages of creating and improving the Monte Carlo AI. In section 4, we will explain all the algorithms we have used for comparison, explain the setup of the experiments and show the results. Then in section 5, we will give our conclusions and give suggestions for future research.

2 The game Ataxx

The name of the game of Ataxx comes from the word 'attack', because each player must play very aggressively to be able to win. Its origin is not known, but it seems to have been developed near 1990. It is also known by other names, like Spot, Infection, Microscope, SlimeWars and Frog Cloning [1, 7].

2.1 Rules of the game

The game Ataxx is played by 2 players, Red and Blue, on a 7×7 game board. In all the figures in this paper, we use white and black stones for visibility. White represents the Red player, Black represents the Blue player. Both players move alternatively. In the starting position (see figure 1), both players have 2 stones. White plays the first move. The goal of the game is to have more stones than your opponent when the game ends.

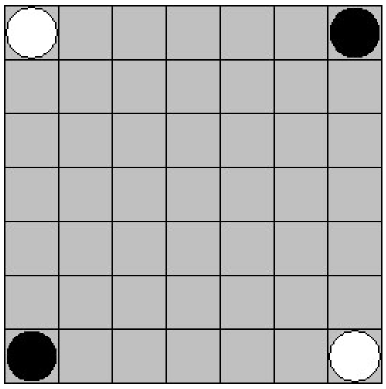

Figure 1: The starting position.

Playing a move

At each turn, a player plays a move if possible. For this, you have two options. You can choose to copy a stone, or to jump with a stone. Both types of moves are only possible when moving to an empty square. When you copy your stone, you can move to all empty adjacent squares (the white squares for the upper left stone in figure 2). When you jump, your stone is not copied, and the stone will be moved to an empty square adjacent to the copying squares (the black squares for the upper left stone in figure 2). When you jump, you can cross a greater distance, but the stone will not copy.

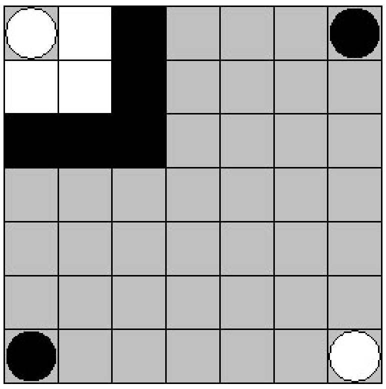
Figure 2: Possible moves for the upper left stone.

Furthermore, with both types of moves, you will take over any stones that are adjacent to the square you are moving to. In other words, all these stones will change to your color. This is illustrated in figures 3 and 4.

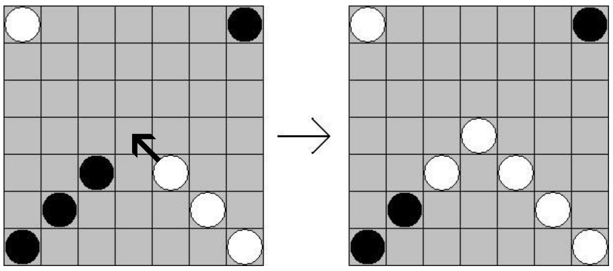
Figure 3: Copying a stone.

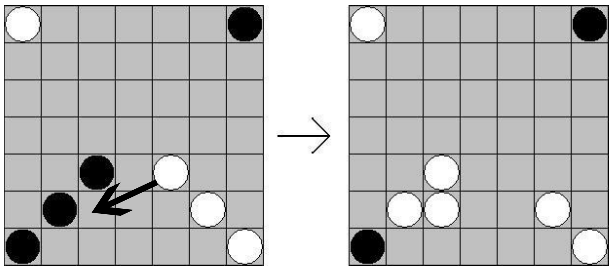
Figure 4: Jumping with a stone.

End of the game

The game ends when all of the 49 squares are filled with stones. The player with the most stones will win the game. The game also ends if one of the players has no stones left. He will lose then.

It can happen that one player has no legal moves, because none of his stones can reach an empty square. If that happens, his turn is skipped and the other player can play another move. This continues until the board is full or until the player gets a legal move again.

Sometimes there is another special rule. In theory, it can happen that the game goes on forever if both players jump all the times and the game gets in some kind of deadlock. There are rules to prevent this. However, there is no fixed rule for the game. In some game implementations, none exists at all. Even though a deadlock does not occur very often, we have encountered some in our tests. Therefore we have implemented the rule that if a board position is encountered for the third time with the same player to move, the game ends and the player with the most stones wins. It can also be a draw if both players have the same amount of stones on the board. It can be argued that declaring a draw when this rule applies (like in chess) is better. We have used the first version of the rule though. This does not influence the test results much, because this situtation occurs very rarely.

3 Implementing Monte Carlo techniques

In this section we discuss the different stages of creating an AI that uses Monte Carlo techniques. The goal for each AI is to select a move as good as possible in a given position, if possible the best move. First, we discuss the basic algorithm. Second, we explain how domain knowledge can be used to increase the performance of the basic algorithm. In the final subsection, we describe a structural enhancement that has been made for even further improvement.

3.1 Basic Monte Carlo algorithm

The basic idea of the Monte Carlo algorithm is to give each possible move in a given position some kind of score or evaluation to indicate the quality of the move. To do this, the Monte Carlo algorithm plays a number of random games for each first move. With a random game we mean that at each turn a random move is chosen from all possible moves. The algorithm plays until the game ends and then evaluates the final score. This is done by an evaluation function, like for example the number of stones or just win, loss or draw. To give a score to each first move possible in the original position, an average of all simulations with that first move is computed. The move with the best average is then chosen.

The choice of the evaluation function can influence the results. If we would choose to evaluate only win, loss or draw, then the weakness is that it does not matter for the function whether you win with an overwhelming majority or wether you win just with a 1-stone difference. This means there will be no distinction between two winning moves, although one can be a very good move and the other just an average move.

Now, if we choose to evaluate the number of own stones minus the number of the opponent's stones, we will have another weakness. Suppose the game is almost finished and still has 1 or 2 empty squares available. If both players are balanced, this function will evaluate a board position with 25 vs 24 stones almost the same as one with 24 vs 25 stones. However, this is the difference between winning and losing the game. In this case the difference between these two should be larger.

Therefore, we will use an evaluation function that uses both of these principles. Points are rewarded for winning, and the more stones you have the more points you will get. The details of the evaluation function we use are discussed in section 4

3.2 Using domain knowledge

The basic Monte Carlo algorithm still plays seemingly randomly and needs improvements. One of the most important issues here is the amount of legal moves that are possible in each position. Most of the times there are more than 50 moves possible (except in the beginning). A game lasts for at least 45 moves if no jump move is executed in the entire game (and if we assume that no player loses all of his stones before the end of the game). In practise, a game will last at least for 50 moves. Even with this lowly estimated amount of moves, the amount of possible games that can be played will be larger than $50^{50} \approx 8.9 \cdot 10^{84}$. A Monte Carlo simulation of 1000 games needs a few seconds to be calculated in the beginning of the game, but 1000 simulations on $8.9 \cdot 10^{84}$ possibilities is just a tiny fraction of all possible games that can be played. Although Monte Carlo always simulates a small fraction of all possibilities, we should improve its efficiency. We do this by decreasing the number of moves that this algorithm looks at to increase the number of simulations done for the best possible moves.

To do this, we will implement domain knowledge into the algorithm. For example, jumping to a square with no surrounding stones is a move you do not want to consider. However, we have to be careful with the process of eliminating 'bad' moves, because they might seem bad at first, but might be very good when you look deeper. Also, we have to find a balance between the amount of heuristics to evaluate a move and the time needed to calculate them. In general, simulation time should be spent on the best moves as much as possible, but also as many simulations as possible have to be done.

Implemented domain knowledge

Here we will explain how we used domain knowledge in the game to improve the Monte Carlo algorithm. To implement knowledge, we will give a score to each possible move. This is done with the following heuristic:

The score S_i for each move m_i starts with value 0, and is modified by the following:
$+s_1$ for each enemy stone taken over
$+s_2$ for each own stone around target
$+s_3$ if move is not a jump move
$-s_4$ for each own stone around the source square (if the move is a jump move)

If $S_i < 0$, then we set $S_i = 0$.

In our program we use $s_1 = 1, s_2 = 0.4, s_3 = 0.7, s_4 = 0.4$. The chosen values have been experimentally determined.

The formula rewards a move that captures a lot of enemy stones. Also, you get points for own stones surrounding the target, since this decreases your vulnerability. Copying is preferred, since this gives you an extra stone. Finally, if a jump move is executed, you get a penalty for each own stone surrounding the origin of the move, since this creates a vulnerable spot.

Note that this heuristic does not evaluate a given position, but only the effect of a given move. The reason for doing this is twofold. First, very little domain knowledge is known and it is therefore not yet possible to write a decent evaluation function that can be used for each and every move. Something like the number of own stones is by far not good enough, since the amount of stones that can be taken over by a single move can be 8. Second, by looking only at the direct impact of a move the calculation time is decreased dramatically. Instead of looking at the entire board (49 positions), you only look at 2 positions (the origin and destination of a move). This means that more simulations can be done.

With the calculated score S_i we can direct the Monte Carlo simulations. Without domain knowledge, each possible move in a given position is chosen with an equal chance (uniform distribution). With the domain knowledge, a move with a larger S_i is chosen more often. The chance $P(m_i)$ for each move m_i to be chosen is:

$$P(m_i) = \frac{S_i^2}{\sum_{j=1}^{M} S_j^2}$$

where $m_1, m_2, ..., m_M$ are all legal moves and $S_1, S_2, ..., S_M$ are their scores. It can happen that a move scores terribly on the heuristic and has a score of 0. If this happens, then $P(m_i) = 0$ and the move is ignored.

3.3 Structural enhancement

As a structural enhancement for the Monte Carlo algorithm we have implemented the idea of playing a tournament of three rounds. In the first round, the simulation is done as usual. This gives a score to each possible move, some of them simulated more often than others. Next, the best 5 moves with the highest score are selected and go to the second round, where more simulations are run on those 5 moves. Finally, the 3 best moves of those 5 moves are chosen for the grand finale, where they are simulated even more. The best of those 3 moves after all simulations are done is chosen as the move to be played.

4 Experiments

In this section we will explain the algorithms and the variations we tested and show the test results. Our main focus will be the Monte Carlo player (MC) and the Monte Carlo player which uses domain knowledge (MCD). They will play against the alpha-beta player (AB) for comparison.

The alpha-beta algorithm ($\alpha\beta$) is one of the basic algorithms used to search a tree in a depth of n. In our case, it searches all possible move variants up to a depth of n and selects the move with the best score. This score is given by an evaluation function that evaluates all tree leaves. If there are multiple moves with the same best score, than the first best move encountered is chosen as the best move. We can also choose to select a random move among the best moves by adding a small random factor to the evaluation. In each position we also sort the possible moves. They are sorted according to the number of stones that are taken over. This sorting decreases the calculation time in some positions with a factor 20. For more information and implementation details on alpha-beta, we refer to [6].

4.1 Set-up

We use the same evaluation function for both $\alpha\beta$ and the Monte Carlo algorithm. For $\alpha\beta$ this evaluation is done for the position after n moves. For the Monte Carlo algorithm, this evaluation is done after the game is finished. The evaluation score E of a position p is the following:

$$E(p) = N_{own} - N_{opp}$$

where N_{own} is the number of own stones and N_{opp} the number of stones from your opponent. If the game is finished, then $E(p) = E(p) + 50$ if you win, or $E(p) = E(p) - 50$ if you lose. If a game is finished before the board is filled, then an extra penalty is given. This means that $E(p) = E(p) + 500$ if you win, or $E(p) = E(p) - 500$ if you lose.

We have a number of options we can vary in our tests with MC. We can choose whether to use domain knowledge and whether to play a tournament. In addition, we can also use the option to vary the amount of simulations depending on the number of empty squares left in the game. If we vary the amount of simulations, the algorithm runs more simulations when the board gets filled with stones. This is possible since the number of moves needed to end the game will be less and more simulations can be done in the same time. The amount of simulations run on a position is $S_{total} = S_{basic} \cdot (1 + 0.1 \cdot n_{filled})$, where S_{basic} is the basic amount of simulations done with an empty board and n_{filled} is the total number of stones on the board.

Finally, we can also vary the number of basic simulations (S_{basic}). You can set the basic amount of simulations for each round of the tournament. A setting with basic simulations 600, 600 and 300 means that 600 basic simulations are done for the first round, divided over all possible moves as explained in subsection 3.2. For the second round, 600 simulations are done for the best 5 moves. Finally, in the last round, 300 simulations are done for the best 3 moves.

We can also choose to vary the amount of moves looked ahead with $\alpha\beta$. This is done because $\alpha\beta$ needs less calculation time in the endgame. $\alpha\beta$ looks 4 moves ahead by default. If we use this option it looks ahead 5 moves when there are 5 empty squares left and 6 moves ahead when there are 2 empty squares left.

4.2 Results

In order to test the performance of the Monte Carlo players, we will let them play against AB and compare the percentage of games won.

One problem arises here. In early tests, it was noticable that it is pointless to test entire games. AB always wins from every variation of Monte Carlo. Even with a lot of simulations it gets beaten easily by AB. The basic MC algorithm is often beaten after only 10 moves. The number of simulations does not even seem to matter much.

If we look at MCD, the results are a bit similar. The same happens as with MC. MCD plays too weak in the beginning of the game and is therefore beaten in every setting we tried. The big difference however is that the MCD plays much better and does not get beaten early on. It plays weak but gets stronger each time the board gets more filled with stones. In the endgame it seems to play pretty strong, but it never wins from AB because it played bad moves in the opening.

Even though MCD cannot win from AB, it is noticeable that it plays stronger in endgames. Therefore, we will test different positions with a varying number of stones on the board (and thus a varying amount of empty squares remaining). We can test when and how MCD can play stronger, because it does not suffer from a bad opening this way. We test this in two different ways. First, we use a combination of an AB player and an MCD player (AB+MCD). This player uses $\alpha\beta$ until there are n_{empty} empty squares left. From that point on in the game, the player will use the Monte Carlo algorithm with domain knowlegde. We vary n_{empty} from 31 to 49. Second, we test with a selection of saved board positions (see appendix A) from real games played by a human player on the internet, with a given number of empty squares.[1] We wil play games with AB+MCD vs AB and vice versa.

MCD vs AB and MCD vs Human

First of all, we try some board positions and use MCD vs AB, where both players play both sides a number of times. We test the same for MCD vs Human. The results are shown in table 4.2. For the positions tested, see appendix A. In this table, H stands for the human player. For the settings, we use tournament and varying amount of simulations (with basic 600 for 1st round, 600 for 2nd round and 300 for the last round). AB looks 4 moves ahead and increases this near the end as described earlier. The AB player does not play a random move yet (from the best moves possible), but selects the first best move it encounters. We let both algorithms play for both sides to get a fair result. With MCD vs AB, 10 games are played for each position, divided equally over both sides. In position 11, MCD played 3 games with White (W) against the human player, and 3 games with Black (B). AB played 2 games with each color against the human player. In position 4, MCD played 3 games with White and 4 games with Black against the human player. The percentages shown in the table for each color indicate the win percentage of the first algorithm with that color against the second algorithm playing the other color.
What we can see so far is that MCD seems to play better than AB in these ending positions. We see a winning rate of 100% for one color and 0% for the other color in most positions. Both algorithms won every game with the same color here, so it seems the position has a clear winner. Noticable are the percentages in position 11 where MCD played against the human player. Here, both players won 1 time with the losing side and made a small error. Also, AB had no chance against the human player in this position, but MCD did. Furthermore, in position 4, MCD has a much higher winning rate than

	Empty	MCD vs AB	MCD vs H	AB vs H
P3:	3	W: 100%	–	–
		B: 0%	–	–
P1:	5	W: 0%	–	–
		B: 100%	–	–
P11:	8	W: 100%	W: 67%	W: 0%
		B: 0%	B: 33%	B: 0%
P4:	11	W: 80%	W: 67%	–
		B: 100%	B: 75%	–
P14:	13	W: 0%	–	–
		B: 100%	–	–

Table 1: Win percentage in different positions with a different amount of empty squares for both algorithms.

AB and almost permanently wins the game with both sides.

Next, we test more board positions. The number of empty squares on the 21 different board positions we used varies from 3 to 39. One important difference is that the AB player now selects a random move from the best moves. MCD uses tournament play and a varying amount of simulations. We test all board positions with 3 different amounts of simulations for MCD. The results obtained are shown in figure 5. In this figure, MCD 300 is the result of MCD with basic simulations 300, 300 and 150. MCD 600 is similar with basic simulations 600, 600 and 300. For each position and both MCD 300 and MCD 600, 10 games are played with MCD vs AB, equally divided over both players.

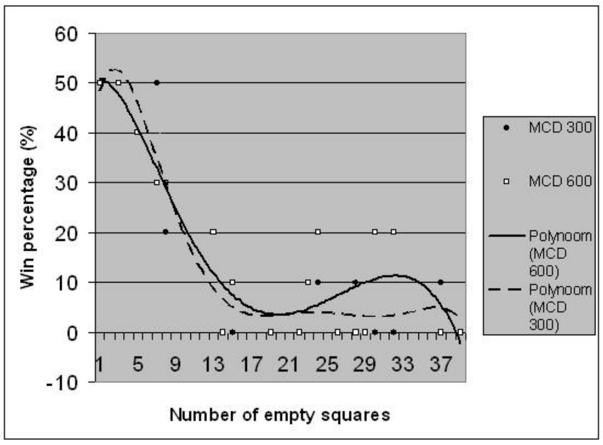

Figure 5: Win percentages of MCD vs AB for different board positions.

In these results we see the effect of the random move used in AB. The positions that were tested earlier where AB did not play a random move yet were a subset of the tested positions here. For all positions that were also

[1]The human player was R. Cuppen, who is a regular player on the internet site www.jijbent.nl with the nickname W84Me2Win. See http://www.jijbent.nl/ratings.php for his Ataxx ranking. Because the game is relatively unknown and there are no clubs for Ataxx, it is hard to indicate how well he plays, but his ranking on the website indicates he certainly plays above average.

tested earlier, MCD has a lower winning rate in these new tests with a random move for AB.

We also see in these results that MCD has a win percentage of about 50 percent in positions with a low amount of empty squares. However, we see a drop in win percentage as the number of empty squares increases. The difference between the amount of simulations is not much, although we notice that MCD 600 wins a bit more often with more empty squares.

AB+MCD vs AB

Next, we test AB+MCD vs AB. We vary n_{empty} from 31 to 49. These results are shown in figure 6. In this figure, MCD 300 is the result of MCD with basic simulations 300, 300 and 150. MCD 600 is the same with basic simulations 600, 600 and 300, and MCD 1200 is the same with basic simulations 1200, 1200 and 600. The calculation time of MCD 1200 is already quite high compared to AB. That is why we do not test with more simulations. For each value of n_{empty} and each of the three simulation settings, 10 games are played with MCD vs AB, equally divided over both players.

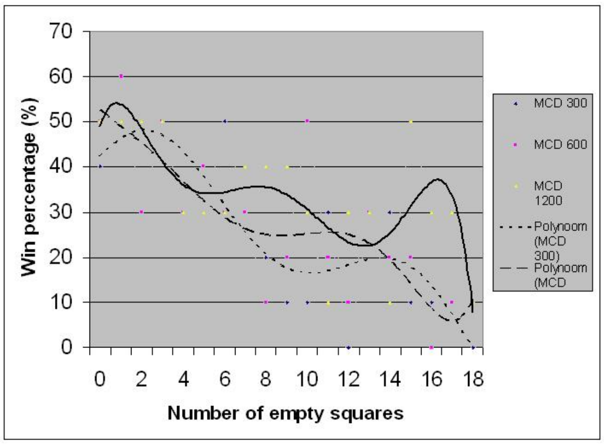

Figure 6: Win percentages of AB+MCD vs AB for different numbers of empty squares (n_{empty}).

Here we see a similar result as with the tested positions. The win percentage is about 50 percent if the switch of AB+MCD from AB to MCD happens later in the game (with less empty fields). This is because with $n_{empty} = 49$, the players are both AB in practise. With $n_{empty} = 48$, only the last or the last few moves are chosen by an MCD player. We see a slow decrease with an increase of empty fields. The more the MCD player plays in the endgame, the lower the win percentage gets.

What we can see clearly in this figure is a higher win percentage for a higher amount of simulations. Especially between $n_{empty} = 30$ and $n_{empty} = 40$ MCD 1200 wins more often than both of the others.

5 Conclusions

In this paper we have implemented and tested a Monte Carlo player for the game Ataxx. We have implemented different levels of enhancement of Monte Carlo. In early tests, it was obvious that the simple Monte Carlo player lost every game against the Alpha-Beta player (AB). Therefore, we compared the best version that uses domain knowledge and tournament play, the Monte Carlo Domain Player (MCD), with AB. Even here, it was obvious that the MCD-player could not win from AB, because it plays bad moves in the opening. Therefore, we tested MCD against AB in real game positions in a later stage of the game. We also combined MCD with AB to test the effectiveness of MCD in later stages of the game. The results showed that MCD plays stronger in the endgame and also a bit stronger with more simulations, but never obtains a win percentage above 50%. It does not play stronger than AB.

This also gives us the answer to our research question: "Can we build a strong AI player for the game Ataxx using Monte Carlo techniques?". MCD has severe weaknesses and is not a very good player in itself. For example, the MCD plays inpredictable, especially in the opening. Sometimes it plays very well, but other times it plays badly. Especially in the opening of the game MCD does not play good moves.

However, in the endgame it shows some stronger points and possibilities. In the endgame less moves are possible and MCD can simulate a larger part of all possible games. Moves are chosen more accurately and less unpredictable, which means the MCD player plays better.

For future research, the MCD player can be improved. One major disadvantage at this moment is the shortage of domain knowledge for this game. With more domain knowledge, the quality of the evaluation of a move or position can be improved and the simulations can be run more accurately and efficiently. It might be possible to decrease the complexity of the game by efficiently eliminating moves that are classified as bad moves. Furthermore, more structural enhancements are possible. In our paper we only implemented tournament play, but there are some more, like Monte Carlo tree search [3] or UCT [5], that can be implemented.

Another option is to do more tests. Testing games with MCD takes much time and due to time restrictions and other circumstances less tests are done than desired. With more testing, more accurate results can be obtained. Variations in the evaluation function and the implemented domain knowledge can be tested to improve MCD. Also, we used only a small amount of test positions. It is possible that these game positions have certain properties which cause MCD to play relatively weak or strong. Using a larger, more representational, test set would give a more accurate indication of the overall game performance of MCD.

With these improvements, a Monte Carlo player might be able to play stronger in the opening and actually become a challenge to the Alpha-Beta player over the entire game.

References

[1] Alain Beyrand (2005). Ataxx !! `www.pressibus.org/ataxx/indexgb.html`.

[2] Bouzy, Bruno (2005). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences, Heuristic Search and Computer Game Playing IV*, Vol. 175(4), pp. 247–257.

[3] G. Chaslot, B. Bouzy J.W.H.M. Uiterwijk, J.-T. Saito and Herik, H.J. van den (2006). Monte-Carlo Strategies for Computer Go. *BNAIC'06: Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence* (eds. W. Vanhoof P.Y. Schobbens and G. Schwanen), pp. 83–90, University of Namur, Namur, Belgium.

[4] IBM (2007). Ibm research. `www.research.ibm.com/deepblue/home/html/b.html`.

[5] Kocsis, L. and Szepesvri, C. (2006). Bandit based Monte-Carlo planning. *European Conference on Machine Learning*, pp. 282–293.

[6] Stuart Russell, Peter Norvig (2003). *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice Hall, New Jersey.

[7] Wikipedia (2007). Wikipedia, the free encyclopedia. `en.wikipedia.org/wiki/Ataxx`.

A Test positions

In this appendix we will give the 22 test positions we used in our tests.

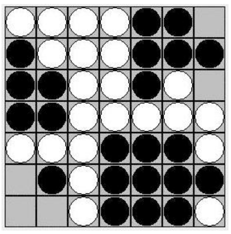
Figure 7: Position 1 with 5 empty sq., White's turn.

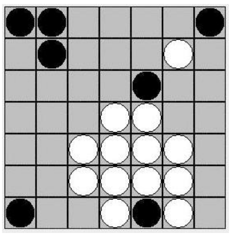
Figure 8: Position 2 with 29 empty sq., Black's turn.

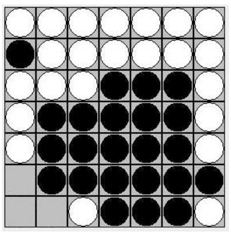
Figure 9: Position 3 with 3 empty sq., White's turn.

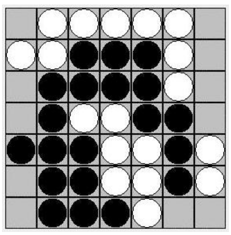
Figure 10: Position 4 with 11 empty sq., White's turn.

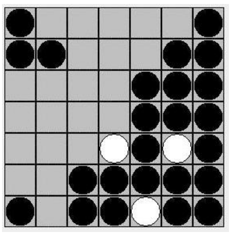
Figure 11: Position 5 with 22 empty sq., White's turn.

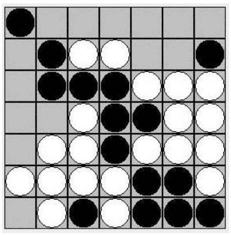
Figure 12: Position 6 with 14 empty sq., Black's turn.

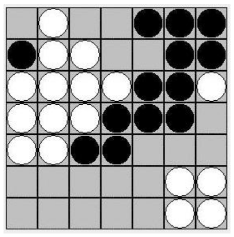
Figure 13: Position 7 with 19 empty sq., Black's turn.

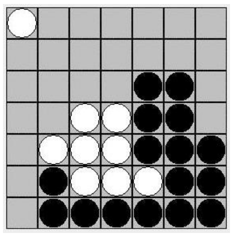
Figure 18: Position 12 with 24 empty sq., White's turn.

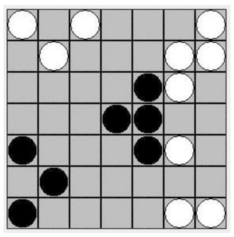
Figure 14: Position 8 with 32 empty sq., Black's turn.

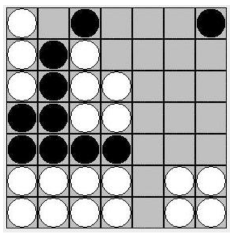
Figure 19: Position 13 with 19 empty sq., Black's turn.

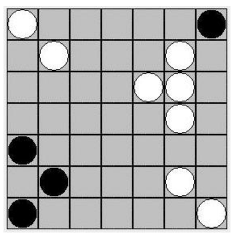
Figure 15: Position 9 with 37 empty sq., Black's turn.

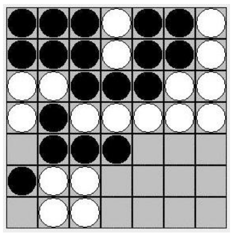
Figure 20: Position 14 with 13 empty sq., Black's turn.

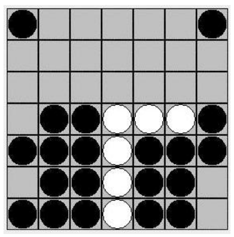
Figure 16: Position 10 with 23 empty sq., White's turn.

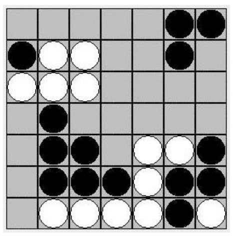
Figure 21: Position 15 with 22 empty sq., White's turn.

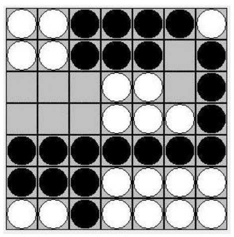
Figure 17: Position 11 with 8 empty sq., White's turn.

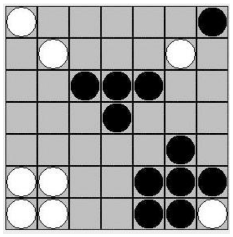
Figure 22: Position 16 with 30 empty sq., White's turn.

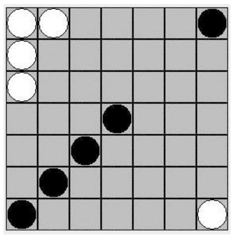
Figure 23: Position 17 with 39 empty sq., White's turn.

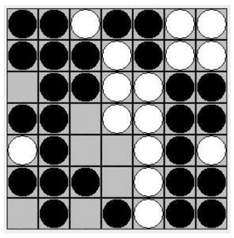
Figure 24: Position 18 with 7 empty sq., White's turn.

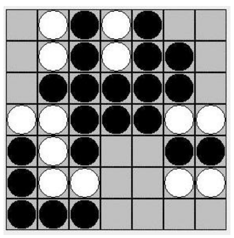
Figure 25: Position 19 with 15 empty sq., White's turn.

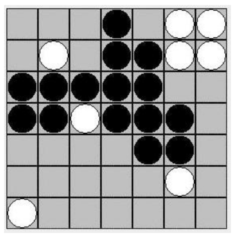
Figure 28: Position 22 with 26 empty sq., White's turn.

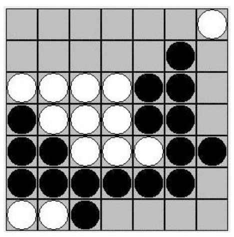
Figure 26: Position 20 with 19 empty sq., White's turn.

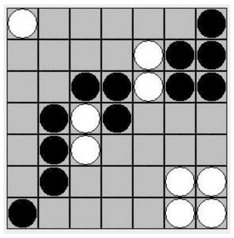
Figure 27: Position 21 with 28 empty sq., White's turn.