Playing Othello Using Monte Carlo

J.A.M. Nijssen

June 22, 2007

Abstract

This paper deals with the construction of an AI player to play the game Othello. A lot of techniques are already known to let AI players play the game Othello. Some of these techniques even allow current AI players to beat even the best human Othello players. These players use very complicated deterministic algorithms to play the game. We will not try to improve on these techniques, but to use a completely different and very simple approach to play the game. This paper shows how a stochastic algorithm, namely the Monte Carlo algorithm, can be used to play the game. It also shows how the algorithm can be improved using domain knowledge and structural enhancements.

1 Introduction

In this section, we will give some background information on Artificial Intelligence, Othello and the Monte Carlo algorithm. Also, the problem statement and the research questions are given. Finally an overview of the paper is presented.

1.1 Background

Artificial Intelligence (AI) is a term that was first introduced in 1956 at the Dartmouth College (see [6], pp. 17-18) and since then it denotes a separate scientific field. The first work in AI however started more than 10 years before the Dartmouth conference and was done by Warren McCulloch and Walter Pitts in 1943 (see [6], pp. 16-17).

One of the first tasks undertaken in AI was game playing. Already in 1950, chess was tackled by Konrad Zuse, Claude Shannon, Norbert Wiener and Alan Turing. In 1956, Arthur Samuel demonstrated a checkers program that was able to play at a strong amateur level, being able to beat its creator.

Nowadays, a lot of AI players have been created for different games that are able to beat even human world champions. Some games are even solved, like *Connect-Four*, *Nine Men's Morris* and *Awari* [8]. One game that is able to be played on a super human level is Othello.[1]

The game Othello was introduced in 1975 [5]. The game is very similar to a game invented around 1880, called Reversi. Reversi was created by Lewis Waterman and John W. Mollett. At the end of the 19th century it gained a lot of popularity in England and in 1898, games publisher Ravensburger started producing the game as one of its first titles [12].

In our research, we developed an AI player that is capable of playing the game Othello. As already explained above, the best AI Othello players are able to beat the human world champions. We did not try to improve on the existing techniques, but investigated a different approach to play Othello. This different approach is the Monte Carlo (MC) algorithm. An advantage of the MC algorithm is that it is fast and does not use any domain knowledge (strategical knowledge about the game). We will explain more about the basic Monte Carlo algorithm in section 3.1.

The MC algorithm has many applications, also in the game field. For example, the algorithm has been used to play the game Go [2]. Go is a very complex game that is very hard to tackle in AI. MoGo, one of the strongest AI Go players, uses the Monte Carlo technique as a basis [9, 10].

1.2 Problem statement and research questions

In this paper, we will discuss how the Monte Carlo algorithm can be used to play Othello, what improvements can be made and how strongly it can play by comparing it to existing AI Othello players. This leads to the following problem statement: "*How can the Monte Carlo search technique be used to play the game Othello, and how does it compare to other methods?*" To answer this question, we define three research questions:

1. What improvements can be made on the Monte Carlo algorithm to make it stronger?
2. How does the algorithm need to be tuned to play as strongly as possible?

[1]Othello is a Registered Trademark, Licensed by Anjar Co., All Rights Reserved

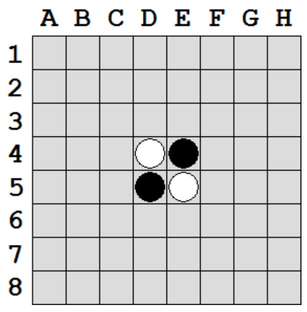

Figure 1: The initial position.

3. How strongly can Othello be played by a Monte Carlo-based AI player?

After showing how the Monte Carlo algorithm is implemented, we will show which improvements can be made. During the experiments, we will show how strongly the MC-based player plays with different settings, by letting it play against a player based on another algorithm. Finally, we will show how a Monte Carlo-based AI Othello player plays against the computer program WZEBRA, which is a strong and popular AI Othello player, and against an experienced human Othello player.

1.3 Overview

In section 2, a description of the game Othello is given. We will give the rules and some well-known strategies. In section 3, we will discuss the workings of the Monte Carlo algorithm and the improvements. Section 4 shows the experiments performed and discusses the results of those experiments. The conclusions that can be drawn from these experimental results are stated in section 5, as well as an outlook to future research.

2 The Game Othello

This section describes the game Othello. First, it gives the rules of the game. Next, it shows some strategies that are commonly known to more advanced players.

2.1 The rules

Othello is played by two players, Black and White, on an 8×8 board. On this board, so-called discs are placed. Discs have two different sides: a black one and a white one. If a disc on the board has its black side flipped up, it is owned by player Black and if it has its white side up, it belongs to player White. The game starts with four discs on the board, as shown in figure 1.

Black always starts the game, and the players make moves alternately. When it is a player's turn he has to place a disc on the board in such a way that he captures at least one of the opponent's discs. A disc is captured when it lies on a straight line between the placed disc and another disc of the player making the move. Such a straight line may not be interrupted by an empty square or a disc of the player making the move. All captured discs are flipped and the turn goes to the other player.

If a player cannot make a legal move, he has to pass. If both players have to pass, the game is over. The player who has the most discs with his color flipped up, wins the game.

2.2 Strategies

Othello is considered to be a game that is 'a minute to learn, a lifetime to master'. This slogan accurately captures the game, because the rules of the game are very simple, but playing the game well is very difficult.

For a computer player, it is important that it has some strategic knowledge about the game. In section 3.2, we will describe how some of this domain knowledge is implemented, but first we describe some commonly known strategies that are used by virtually every advanced Othello player.

Corners and stable discs

During a game some discs, especially those in the middle, are flipped a lot of times. Discs that still can be flipped during the current game are called unstable discs. Stable discs, on the other hand, cannot be flipped anymore during the current game. This means that players owning stable discs cannot lose them anymore.

Discs placed in the corners of the board are always stable. Discs that are adjacent to stable discs can also be stable. This makes corners strategically very important. In the example in Figure 2, the dark shaded squares indicate the stable discs for Black.

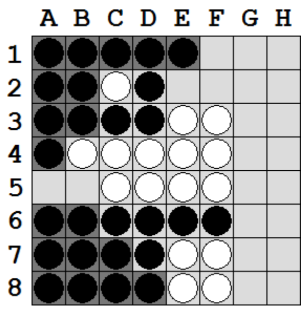

Figure 2: Stable discs for Black.

Location

An Othello board consists of 64 squares. Strategically, each of those squares has a certain importance. As explained above, the four corner squares are very important.

To make strategic discussions easier, most squares on an Othello board are given names. Besides the corners, an Othello board contains X-, A-, B-, and C-squares and the Sweet 16. In figure 3, the locations of these squares are shown. The 16 dark-shaded squares in the middle of the board belong to the Sweet 16.

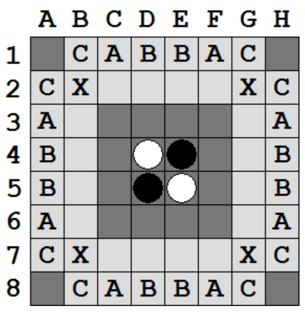

Figure 3: Types of squares on an Othello board.

Typically, playing X-squares or, to a lesser extent, C-squares are considered to be strategically bad moves. If a player owns such a square, he risks losing the corner.

Placing a disc in the Sweet 16, however, often does not bear any serious risks and is often a good move.

Mobility and frontiers

Each player is obliged, if possible, to play a move. Because of this rule, it is possible to force the opponent to make bad moves. This can be achieved by limiting the number of moves the opponent can do. Advanced players can place their discs in such a way to limit the possible moves of the opponent to only bad ones.

One way to increase your own and limit the opponent's mobility is to make your frontier as small as possible. All discs that are adjacent to at least one empty square belong to the frontier. A large frontier often means a large number of possible moves for the opponent, whereas a small frontier limits this number.

Parity

Playing the last move in a game is generally a major advantage. Discs that are flipped during the last move cannot be lost anymore and this feature gives the player doing the last move a considerable advantage. Typically, because Black begins the game, White can make the last move. However, when someone has to pass, the parity changes from one player to the other. So, in order to make the last move, Black should try to force an odd number of passes in the game to 'win parity'. White, on the other hand, tries to either avoid any passes or to force an even number of passes in the game.

3 The Monte Carlo Algorithm

This section describes the Monte Carlo algorithm. First, it shows how the basic algorithm works. Second, we show how the algorithm can be improved by applying domain knowledge. Third, a possible structural enhancement to the algorithm is described.

3.1 The basic algorithm

When a player wants to make a move, he should first know which moves are legal. These moves are called the candidate moves. If there are no candidate moves, then he has to pass. If there is only one candidate move, then this move is by definition the best one. If there is more than one candidate move, he has to make a decision. This decision is made by the Monte Carlo algorithm.

The algorithm is run for a fixed number of times. This number is indicated by the variable N_p. These runs are equally divided among the candidate moves. During each run, the algorithm first plays the candidate move and then continues playing the game completely randomly, until the game is over.

When all runs of a candidate move are done, the average result is computed. This average result can be computed in two ways: either by calculating the percentage of won games, or by calculating the average number of discs owned by the player at the end of a game. In our algorithm, we use percentage of won games, as it is more important to try to win the game than to try to have on average many discs. The average result gives an indication of whether the candidate move was a good one or not.

When all runs are done, the candidate move with the highest average result is considered to be the best move and is eventually played.

3.2 Applying domain knowledge

The basic Monte Carlo algorithm does not use any domain knowledge at all. The advantage of this is that the algorithm is very easy to implement and works faster than algorithms that do use domain knowledge. The disadvantage is that, because of the lack of strategic knowledge, the algorithm does not play very strongly. This will be explained further in section 4.

To improve the algorithm, domain knowledge can be applied in two ways. The first one is to use domain knowledge to create a preprocessor. The second is to incorporate domain knowledge into the algorithm to 'steer the algorithm in the right direction'. This technique is called pseudo-random move generation [1].

Preprocessing

In the basic Monte Carlo algorithm, all candidate moves are played an equal number of times, including the strategically bad ones. The preprocessor analyzes all candidate moves and rewards each move with a score. The higher the score, the better the move. This score is based on the strategic points explained in section 2.2. There are two types of preprocessors.

The first one selects a fixed number of moves. We call this preprocessor the Fixed Selectivity Preprocessor (FSP). After the scoring procedure, the FSP selects the N_s moves with the highest score. N_s is a parameter that can be tuned (see section 4). These moves are then passed on to the Monte Carlo algorithm. The other moves are eliminated and are not considered anymore. This way, the strategically bad moves are filtered out and only the strategically most promising moves are considered.

The second type of preprocessor selects a variable number of moves. This preprocessor is called the Variable Selectivity Preprocessor (VSP). The VSP selects moves by comparing their scores to the average score of all candidate moves. If the score of a candidate move is at least a given percentage, called p_s, of the average score, it passes the preprocessor. Otherwise, it is filtered out. By lowering the value of p_s, more moves pass the preprocessor.

Pseudo-random move generation
In the basic Monte Carlo algorithm, after playing the first move, the game is finished by playing each move with an equal probability. This means that the moves are drawn from a uniform distribution.

When applying pseudo-random move generation, moves are not drawn from a uniform distribution, but they are chosen with a given preference. This preference is determined by the domain knowledge. Each move is scored in the same way as with the preprocessor. The higher the score, the more the move is preferred to be played. The advantage of this is that the more likely game progressions are played more often than the less likely ones. The disadvantage is that computing the scores for each move in each game takes a lot of time.

In order to limit the time spent on scoring the moves, the game is not completely played using pseudo-random move generation, but only during the first N_d moves. After that, the moves are drawn using the uniform distribution. The larger the value of N_d, the more time is spent on scoring, meaning that less games can be played. In section 4, results for different values of N_d are shown.

3.3 Implementing structural enhancements

The Monte Carlo algorithm can be improved by adding domain knowledge, but it is also possible to apply structural enhancements. Such enhancements make the algorithm work more efficiently, without adding any domain knowledge to the algorithm. One of these techniques, called tournament play, is explained below.

Tournament play
As explained in section 3.1, all runs are equally divided among the candidate moves. In order to improve the accuracy of the algorithm, tournaments can be played. First, the candidate moves are being played a number of times. After analysis, the move with the lowest average result is eliminated and the remaining moves are being played again a number of times. This process is repeated until two candidate moves are left and the best one is chosen. The advantage is that the last moves are played more often, meaning that their results are more accurate. A possible downside is that moves that are eliminated during the first rounds might not have been evaluated often enough to be sure that they really are bad moves.

Square type	Owner	
	alpha-beta	opponent
Corner	+5	-5
X-square	-2	+2
C-square	-1	+1
Sweet 16	+2	-2
Others	+1	-1

Table 1: Score awarded to a disc based on its owner and location.

4 Experiments

In order to answer the second and the third research question stated in the introduction, we needed to execute a number of experiments. Based on these experiments, we can define the settings for the Monte Carlo algorithm and its improvements to play as strongly as possible. In this section, we will show how the experiments are set up and what the outcomes of the experiments are. At the end of this section, we will show how the MC algorithm plays against other Othello players.

The conclusions that can be derived from these experimental results are described in section 5.

4.1 The alpha-beta player

In order to see how strongly the MC-based player plays, we decided to let it play against an AI player based on another algorithm, namely the alpha-beta algorithm. This algorithm is easy to implement and is able to play the game pretty well, even without too much domain knowledge.

The depth of the search tree of the alpha-beta algorithm is, during our research, limited to 7. This lets the algorithm search deep enough to play a decent game, without having a too large computing time.

At the leaves of the search tree at depth 7, the board is evaluated and returns a score. The alpha-beta player tries to maximize this score. Each disc on the board is awarded a score. This score is based on the owner of the disc and its location on the board. Table 1 shows how the scores for each disc are determined.

N_p	wins (%)	average score	average computing time (ms)
0	0	4.65	0.74
100	17	14.52	21.78
250	23	17.84	49.70
500	27	19.06	107.47
1000	36	22.43	208.21
2500	40	25.22	515.22
5000	51	26.25	1017.08
10000	52	28.09	2037.96

Table 2: Experimental results for different values of N_p without a preprocessor and $N_d = 0$.

If at a leaf the game is finished, 1000 points are awarded if the alpha-beta player has won, and -1000 points if it has lost.

4.2 Variable tweaking

As explained in section 3, the Monte Carlo algorithm contains a number of variables that can be tuned to change the way the algorithm works. To summarize, there are three variables that can be set: the number of games played (N_p), the maximum number of moves that are allowed to pass the preprocessor (N_s or p_s, based on the type of preprocessor) and the maximum depth at which domain knowledge is used to steer the Monte Carlo algorithm (N_d). Besides these three variables, the use of tournament play can be toggled on or off. This section will show how the MC player plays against the alpha-beta player described above with different variable settings.

Number of plays

The Monte Carlo algorithm plays each move a given number of times. The larger the number of plays, the more reliable the results will be. But simulating more games also takes a longer time. The first set of experiments is conducted to investigate the influence of the number of games that are simulated on the strength of the player. We let the MC player play 100 games for each setting against the alpha-beta player. The results for different values of N_p against the alpha-beta player are displayed in Table 2.

For these experiments, we did not use a preprocessor and we set $N_d = 0$, meaning that no domain knowledge is used in the simulations of the Monte Carlo algorithm. Note that for $N_p = 0$, no games are simulated, so the algorithm chooses one move completely randomly.

Preprocessor

In the second set of experiments, the influence of the preprocessor on the performance of the MC algorithm is investigated.

N_s	wins (%)	average score	average computing time (ms)
2	45	26.25	535.33
3	40	25.46	515.60
4	51	29.08	511.13
5	48	27.31	526.17
6	41	22.49	517.65
7	42	22.39	537.70
8	45	24.88	537.22
9	46	26.35	539.20
10	40	24.21	542.33

Table 3: Experimental results for the FSP for different values of N_s with $N_p = 2500$ and $N_d = 0$.

p_s	wins (%)	average score	average computing time (ms)
100%	32	22.33	488.89
90%	34	23.64	492.21
80%	48	25.53	488.22
70%	52	27.88	482.56
60%	43	25.41	493.44
50%	47	26.46	508.93
40%	47	27.14	476.27
30%	44	25.17	500.51
20%	38	23.11	489.88
10%	40	24.89	493.61

Table 4: Experimental results for the VSP for different values of p_s with $N_p = 2500$ and $N_d = 0$.

In this set of experiments, we set the number of simulated games of the Monte Carlo algorithm to a fixed number, namely $N_p = 2500$. During the simulation process of the Monte Carlo algorithm, no domain knowledge is used, so $N_d = 0$. The only domain knowledge is used in the preprocessor. For each setting, again 100 games are played against the alpha-beta player with a maximum depth of 7. As explained in section 3.2, we can use two different preprocessors.

Table 3 shows the experimental results for the FSP for various values of N_s. Please note that for $N_s = 1$ no experiments are conducted. This is because if $N_s = 1$, only one move passes the preprocessor, meaning that the Monte Carlo algorithm is not used to determine which move is played.

In Table 4 the experimental results for the VSP are displayed for various values of p_s.

Domain knowledge in the Monte Carlo algorithm

As explained in section 3.2, domain knowledge can be used to give strategically better moves a higher preference to be played during the Monte Carlo simulation. In the third set of experiments, we investigate the influ-

N_d	wins (%)	average score	average computing time (ms)
0	40	25.22	515.22
2	51	27.49	755.35
5	49	26.67	1414.57
7	51	27.66	1887.92
10	55	28.90	2504.83
12	62	29.77	2959.58
15	55	27.83	3585.34
17	52	27.94	4057.45
20	60	30.44	4666.17

Table 5: Experimental results for different values of N_d with $N_p = 2500$ and $N_s = 99$.

N_p	wins (%)	average score	average computing time (ms)
0	0	4.65	0.74
100	14	14.66	18.50
250	24	18.61	47.85
500	38	23.03	97.08
1000	36	23.68	193.72
2500	46	26.72	478.88
5000	45	24.27	964.54
10000	50	27.89	1895.88

Table 6: Experimental results for different values of N_p using tournament play, with $N_s = 99$ and $N_d = 0$.

ence of the amount of domain knowledge on the performance of the algorithm. During these experiments, we let the value of N_d vary to discover the effects of domain knowledge when applied in the Monte Carlo algorithm on its overall performance. As in the second set of experiments, we use $N_p = 2500$. To fully observe the effect of domain knowledge used in the MC algorithm, we disable the preprocessor. Again, we let the MC player play 100 games at each setting against alpha-beta with a maximum depth of 7. In Table 5, the results for different values of N_d are displayed.

Tournament play

If tournaments are used, bad moves are eliminated early on, while good moves are played more often. In this set of experiments, we use the same setup as in the first set of experiments, but now with tournaments instead of without. Table 6 shows the results retrieved from these experiments.

4.3 Performance against other players

In the second part of the problem statement, we ask how strongly the algorithm plays against other players. Therefore, we have created a Monte Carlo-based AI Othello player, named MONTHELLO. For MONTHELLO, we set $N_p = 5000$ and $N_d = 10$. We chose to use the FSP and set $N_s = 5$. Finally, MONTHELLO does use tournament play. In section 5.1, we will show that these settings will provide the best results.

Opponent	plays	wins	average score
Alpha-beta	100	62	31.25
WZEBRA (depth = 1)	10	6	29.60
WZEBRA (depth = 4)	10	0	10.60
Human	10	1	15.30

Table 7: Results of MONTHELLO against various players.

There are three opponents for MONTHELLO to play against: the alpha-beta player, which was used to tweak the variables, WZEBRA, a strong and popular AI player, and an experienced human player, namely the developer of the program and the writer of this paper.

First, we let MONTHELLO play against the alpha-beta player. It turns out that MONTHELLO is able to win the majority of the games. Out of 100 games, MONTHELLO wins 62 times and loses 38 times.

Second, we let our program play against WZEBRA. It is possible to manually set the strength of this program. We let our program play against two different levels. First, we set the strength as low as possible, with a search depth of 1, no use of opening books and small randomness during the mid game. With these settings, MONTHELLO manages to win 6 out of 10 games. If we increase the strength of WZEBRA by setting the search depth to 4 and toggling the use of opening books on, WZEBRA easily wins all 10 games.

Finally, we let MONTHELLO play against an experienced human player. Out of 10 games, MONTHELLO manages to win one of them in a pretty strong game. It loses all nine other games, however.

The results are summarized in Table 7.

5 Conclusions

In the introductory chapter of this paper, we have given a problem statement and three research questions. In this section, we will summarize some key points of this paper while answering the research questions. After that, we will use these answers to give a concluding answer to the problem statement. Finally, we will give some suggestions for future research.

5.1 Answering the research questions

In this section, we will give an answer to the research questions.

Improvements

The first research question was how the MC algorithm could be improved.

In section 3, we have proposed some improvements. Domain knowledge can be used as a preprocessor to filter out bad candidate moves and only consider the best ones, and as a basis for a pseudo-random move generator, where, during the simulations, better moves have a higher priority to be played than bad moves. One way to structurally improve the algorithm is by implementing tournaments, so that better candidate moves are played more often than bad ones.

Tuning

The second research question was how the parameters need to be tuned to make the algorithm work as strong as possible.

As the results in Table 2 show, the strength of the algorithm strongly depends on the number of games that are simulated. Starting at $N_p = 0$, where no games are won, the strength of the algorithm quickly increases as N_p increases. However, starting at $N_p = 5000$, the strength algorithm seems to stabilize, so it seems unnecessary to simulate more games, because they do not really improve the strength of the algorithm, while they do increase the computing time considerably. So, $N_s = 5000$ seems to give the best balance between performance and speed.

Table 3 shows the experimental results for the FSP for various values of N_s. The best results are achieved with $N_s = 4$ and $N_s = 5$. Apparently, if N_s is lower than 4, the preprocessor sometimes filters out good moves. This is because the evaluation function does not reward those moves with an appropriate score. We will explain more about this in section 5.3. If N_s is larger than 5, the performance is also lower. This is because more moves pass the preprocessor, including bad ones. The higher N_s, the more often it happens that the preprocessor does not filter out any move, making the effect of the preprocessor smaller. The fluctuation of the win rate can be attributed to the randomness of the Monte Carlo algorithm.

Basically, the same goes for the results for the VSP. As can be seen in Table 4, if the preprocessor is too strict, good moves are filtered out too often. If the value of p_s drops, however, the effect of the preprocessor diminishes because bad moves also often pass the preprocessor. The VSP seems gives the best results for $p_s = 70\%$.

Both preprocessors seem to be equally strong if their parameters are set optimally.

Table 5 shows the influence of domain knowledge on the performance of the MC algorithm. It shows that even a little domain knowledge ($N_d = 2$) considerably improves the performance of the algorithm. Adding domain knowledge to a deeper level improves the algorithm, but it also makes it significantly slower. Adding domain knowledge up to a depth of 20 makes the algorithm about ten times as slow as without domain knowledge. So, domain knowledge effectively increases the strength of the algorithm, but it is also very time consuming. Because of the fluctuation in the results, it is hard to determine the best value for N_d. It is best to set N_d not too high, because the processing time greatly increases if N_d becomes larger. $N_d = 10$ adds enough domain knowledge to the MC algorithm to make it work better, while the computation time is not too large.

The results of the fourth set of experiments, which investigates how the performance of the MC algorithm changes if tournament play is applied, are shown in Table 6. To see how the tournaments influence the average outcome, the results need to be compared with those in Table 2. As these tables show, adding tournaments does not significantly improve the algorithm. If N_p gets larger, tournament play makes the algorithm work a little bit better. The only significant different occurs for $N_p = 500$. In this case, the algorithm is able to work pretty well already without tournaments. However, if tournament play is used, the number of simulations during the first rounds is way too small. This means that it happens often that good moves are eliminated during the first rounds.

Strength

From the final set experiments, we can conclude that Monthello is able to play well against simple AI players, but it barely stands a chance against an experienced human player or a strong AI player.

Evaluation of the played games shows that one of the biggest problems is still that Monthello sometimes plays a really bad move. Figure 4 shows one of those moves. Monthello, playing black, plays on square H2, giving away corner H1. During the simulation, however, Monthello wins the majority of the games, because if the games are solved pseudo-randomly, it happens more often that Black takes corner H8 than that White takes corner H1. This is because Black can make less moves than White, and therefore has a higher chance of taking the corner.

5.2 Answering the problem statement

In the introduction we asked how the Monte Carlo algorithm can be used to play the game Othello and how strongly it plays. In this paper, we have shown that the basic MC algorithm is sufficient to play the game. The performance, however, is poor. Therefore, improvements are absolutely necessary to let the algorithm play at a higher level. In the previous section, we have shown what improvements can be made and how they need to be tuned to play as good as possible, while still being fast.

Monthello turns out to be a decent Othello player. Sometimes it gives away a game because of a bad move.

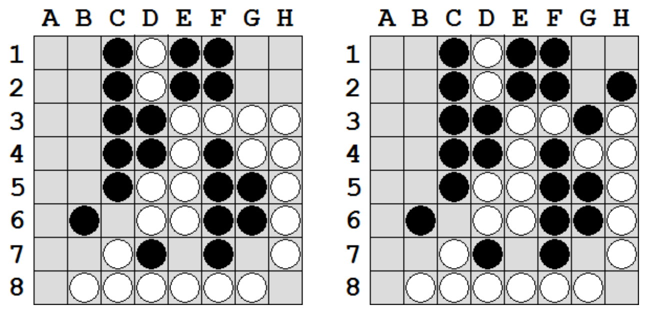

Figure 4: A situation which occurred during one of the games against the human player. MONTHELLO (black) plays H2.

Domain knowledge helps to minimize the amount of bad moves, but they still occur because of the stochastic nature of the Monte Carlo algorithm. Still, for experienced players and strong AI players it is not too difficult to beat MONTHELLO, even if it does not make any bad moves.

Our research shows that the Monte Carlo algorithm can also be used to play other games than Go. As indicated in the introduction of this paper, MoGo is one of the strongest AI Go players around. However, MoGo currently plays at a strength of 4 kyu [11], which stands for an intermediate amateur. Also MONTHELLO does not seem to be able to play at a level higher than amateur. There are some improvements that can be made to MONTHELLO, which will be discussed in section 5.3, and while they will make the program stronger, it is not likely that they will make the program play at world champion level.

It turns out that Monte Carlo is a simple algorithm that can be used to play strategically complex games pretty well, but not on a very high level.

5.3 Future work

In our research, we incorporated a couple of improvements to make the algorithm work better. Naturally, there are many more improvements that could be made that were not covered in our research.

For instance, one way to structurally improve the Monte Carlo algorithm is the implementation of Monte Carlo Tree Search [3]. Another is the implementation of Upper Confidence bounds applied to Trees (UCT). UCT is an often-used technique to structurally improve the MC algorithm without adding new domain knowledge [4]. Starting from the current state of the game, UCT selects a path of moves through a search tree by computing a value for each candidate move based on the average outcome of the move and how many times the move has been played, as well as how many times the position has been visited. Based on this value, each move is assigned a preference to be played. If there are children to a node that have not been visited yet, then one of those moves is selected at random [7].

Another way to improve the performance of the algorithm is improving the domain knowledge. For instance, sometimes the preprocessor filters out good moves because these moves were not rewarded appropriately by the evaluation function. Making a good evaluation function for Othello is difficult, because it is a strategically very difficult game. Some easier things however could be implemented in the future, like detecting wedges (placing a disc on an edge between two opponent discs).

The algorithm could also be improved by learning common openings. This way, for instance bad moves in the early game can be eliminated. A good opening also gives a player a considerable advantage in the rest of the game.

References

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

[2] Brugmann, B. (1993). Monte Carlo Go.

[3] Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2006). Monte-Carlo Strategies for Computer Go. *BNAIC06: Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, pp. 83–90.

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

[5] Othello University (2007). Othello history. `http://home.nc.rr.com/othello/history/`.

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

[7] Sensei's Library (2007). UCT for Monte Carlo computer Go. `http://senseis.xmp.net/?UCT`.

[8] Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games solved: Now and in the future. *Artificial Intelligence*, Vol. 134, pp. 277–311.

[9] Wedd, N. (2007a). Computer Go - past events. `http://www.computer-go.info/events/index.html`.

[10] Wedd, N. (2007b). Detais of program: Mogo. `http://www.computer-go.info/db/oprog.php?a=Mogo`.

[11] Wikipedia (2007a). Computer Go. `http://en.wikipedia.org/wiki/Computer_Go`.

[12] Wikipedia (2007b). Reversi. `http://en.wikipedia.org/wiki/Reversi`.