Using Intelligent Search Techniques to Play the Game Khet

J. A. M. Nijssen J. W. H. M. Uiterwijk

Maastricht University, P.O. Box 616, 6200 MD Maastricht

Abstract

This paper describes the analysis of the game Khet and the implementation of a game engine. Both the state-space complexity and the game-tree complexity of Khet are given. They turn out to be of the same order as those of chess. Based on these results, search techniques are selected that can be used to create an AI player which can play Khet as good as possible. These search techniques are improved to make them run more efficiently, so they can search more extensively and, thus, play stronger.

From the experiments that are performed, we can conclude that Khet can best be played by the alpha-beta search algorithm, enhanced by a transposition table, killer moves, and Q_n-limited search, a variation on quiescence search. These three enhancements yield a significant improvement of the strength of the AI player. Using these search techniques we are able to create a computer program which can play Khet at a strong amateur level.

1 Introduction

Since the advent of computers, board games have produced a new challenge: to let computers play the games that humans have been playing for centuries. The first challenge is to make a computer player for a complex strategic game that is better than any human in the world. The ultimate challenge is to solve the game, so a computer player can be created that always plays optimally. The idea of having computers playing games has been around for over 50 years already. In the early 1950s, Shannon [12] and Turing [13] were the first researchers to describe how computers can be used to play chess. This would eventually lead to the creation of the DEEP BLUE chess computer which defeated the world champion, Garry Kasparov, in 1997 [7].

Nowadays, still many new board games are invented. One game that combines the simplicity and abstractness of classic board games with modern techniques is called *Khet*. Khet was invented in 2004 under the name *Deflexion*. It can be described as a combination of chess, checkers and lasergaming. The rules are simple, but it requires a lot of strategic thinking in order to play it well. Since Khet is still such a new game, only little information can be found about it. No research in the field of Artificial Intelligence concerning Khet has been done so far. The goal of this research is to make a computer program that is able to play Khet as good as possible. This paper describes which search techniques are used, how they are improved, and how they perform compared to each other.

The remainder of this paper is structured as follows. Section 2 gives an introduction to Khet. Also, the state-space complexity and the game-tree complexity of Khet are given. Section 3 gives a description of the search techniques that are used, including the improvements that are implemented. Section 4 gives a description of the experiments that have been performed and shows the results. Finally, Section 5 lists the conclusions that can be drawn from the experimental results. It also describes possible future research subjects, which can be investigated in order to improve the strength of the Khet-playing program.

2 The Game Khet

Khet is a new game, formerly known as Deflexion. The game was invented by Tulane University professor Michael Larson and two students, Del Segura and Luke Hooper. Deflexion was first released in 2004. In 2006, its name was changed to Khet. Currently, the game is released by Innovention Toys LLC.

2.1 Rules

Khet is a 2-player zero-sum game with perfect information. The two players are called Silver and Red. It is played on an 8×10 board. Silver always starts.

Each player receives 14 pieces at the start of the game: 7 pyramids, 4 obelisks, 2 djeds and 1 pharaoh. Some of these pieces have mirrors attached to them. Obelisks and pharaohs have no mirrors. It is possible to stack two obelisks of the same color on top of each other, resulting in a stacked obelisk. The advantage of stacking is that the two obelisks can be moved in one turn. Pyramids have one mirror and two unprotected sides. Djeds have two mirrors. Since they have no unprotected sides, they cannot be captured and thus are always present on the board.

The two players take turns alternately. Each move consists of two parts. First, the player needs to move one of his pieces. This can be either by moving a piece one square orthogonally or diagonally, or by rotating it by 90 degrees, either clockwise or counter-clockwise. A piece can only be moved to an adjacent square if the target square is not occupied by another piece. After moving one of his pieces, the player has to fire his laser, which is mounted on a fixed location inside the edge of the board. If the laser beam hits a piece on a mirrored side, it is reflected with an angle of 90 degrees. Eventually, the beam will either hit the side of the board, or it will hit a piece on an unmirrored side. If a piece is hit on an unmirrored side, it is captured and removed from the board, otherwise nothing happens. After the laser has been fired and, if necessary, the captured piece has been removed from the board, the other player is to move.

The game is over whenever one of the pharaohs is captured. The player with the remaining pharaoh is the winner of the game. The game can also end in a draw. Whenever the same board position occurs for the third time, the game is drawn. Two board positions are the same when the same type of pieces with the same colors occupy the same squares with the same orientation, and the same player is to move.

For more information about the game and the rules, see `http://www.khet.com`.

2.2 Complexity

In order to get some insight into the game, the complexity of the game has to be determined. There are two types of complexity measures: the state-space complexity and the game-tree complexity. For a full complexity analysis of Khet, see [11].

The state-space complexity denotes the number of possible states a game can be in. If the number of possible states is small enough, it is possible to store the best move for a player for each state. However, for most games, including Khet, the number of states is too large. A Khet board consists of 80 squares: 10 silver ones, 10 red ones and 60 neutral ones. Note that silver pieces may not be placed on red squares, and vice versa. Using these data, we can compute that the number of possible states with all pieces on the board is roughly 10^{48}. This number does not include the states where one or more pieces have been captured. Calculations show, however, that if there are less pieces on the board, the number of possible states drops significantly. This means that any further calculations are unnecessary and that the state-space complexity of Khet can be estimated as 10^{49}.

The game-tree complexity indicates the number of terminal nodes in a complete game tree. In other words, it denotes how many different games can be played. The game-tree complexity can be computed as b^d. Here, b denotes the branching factor of the search tree and d the depth of the tree. In other words, b is the average number of valid moves for a player and d is the average length of a game. Unfortunately, there is no information available on the characteristics that are used to determine the game-tree complexity. We have to use the computer program to determine the average braching factor and the average game length. From the experiments, we can conclude that the average branching factor in Khet is 69 and the average game length is 68. This leads to a game-tree complexity of 10^{125}.

To put these numbers into perspective, we need to compare them to those of other games. It turns out that both complexity measures of Khet are comparable to those of chess, which has a state-space complexity of 10^{46} and a game-tree complexity of 10^{123} [6]. This means that Khet is most likely unsolvable, at least in the near future. Because of the large state-space complexity, it is infeasible to enumerate all possible states. The large game-tree complexity makes it impossible to perform a full-depth search in the game tree of Khet.

3 Search Techniques

In this section, we give a description of the tree-search techniques that are used to let the computer program play Khet. There are two search techniques that we investigate: alpha-beta search and Monte-Carlo Tree Search (MCTS). Alpha-beta search is the most commonly used search technique in games. MCTS is a technique that is not used as often as alpha-beta search, but it has given good results in a number of games. The best known example is *Go*. MCTS has some advantages over alpha-beta search that could make it a good alternative.

3.1 Alpha-beta search

The alpha-beta search technique [8] is basically an improved version of the minimax search algorithm. During the traversal of the search tree, the algorithm might stumble upon moves that can impossibly affect the outcome of the search. The alpha-beta search algorithm takes advantage of this by pruning subtrees that cannot affect the result.

3.1.1 Evaluation function

At each leaf node of the alpha-beta search tree, the evaluation function is called to give a score to the board position. The evaluation function is absolutely the most important part of the alpha-beta search algorithm.

Since Khet is still a new game, there is not much information about strategies available. The implemented evaluation function only considers the most basic strategic elements. The most important element is the number of pieces. Usually, having more pieces than the opponent is better than having less. The player with more pieces has more control over the board. To implement this in the evaluation function, each piece is given a certain value. If a piece belongs to the player, the value is added to the total, if the piece belongs to the opponent, the value is substracted. Table 1 lists the values of all pieces, along with some remarks.

Piece	Value	Remarks
Obelisk	10,000	Value is divided by the Manhattan distance to the pharaoh
Stacked obelisk	25,000	Value is divided by the Manhattan distance to the pharaoh
Pyramid	75,000	5,000 points extra if it sends the laser beam into the playing field
Djed	0	Always present, so the values always cancel out
Pharaoh	0	Always present, so the values always cancel out

Table 1: Values of the Khet pieces.

Finally, a random value is added to the score. This is a 12-bit value, so it lies between 0 and 4,095. This random value is added to ensure that the computer player can play different moves in identical situations. Also, by introducing a random factor, mobility is implicitly rewarded. As a consequence, if a move leads to more good moves for the player and less good moves for the opponent, it has a higher chance of being played.

3.1.2 Improvements

The alpha-beta search algorithm can be improved in numerous ways. In our program, we implemented the following improvements: incremental move generation, quiescence search, transposition tables, killer moves, aspiration search and avoiding self-destruction.

Usually, at each node in the search tree the complete set of valid moves for a player is generated. Whenever pruning occurs, all remaining moves are left uninvestigated, but they still were generated by the move generator, which is a waste of time. Incremental move generation only generates the moves for one piece at a time, which means that the moves for the pieces that are uninvestigated whenever pruning occurs are not generated, which saves time.

The second improvement is quiescence search [1]. Quiescence search can be used to take care of the horizon effect. At each leaf node, all moves that cause a piece to be captured are further investigated. For each of these resulting board positions, this process is repeated, until there are no more capture moves left.

For the basic quiescence-search algorithm, there is no limit on how deep the search can go. As long as capture moves exist, the search tree can grow deeper. Since it is possible in Khet to create long sequences of capture moves, it can be a good idea to limit the search depth of the quiescence search. With Q_n-limited search, at each leaf node of the regular alpha-beta search tree, quiescence search can build a subtree with a maximum depth of n. All nodes at depth n of the quiescence-search tree are leaf nodes. When applying Q_n-limited search, the horizon effect still exists. In order to minimise this effect, we only allow leaf nodes to occur after a sequence of an even number of capture moves, so both players have the same amount of capture moves. This means that at depth $n - 1$ in the quiescence search tree, nodes that follow a sequence of an even number of capture moves are not expanded, since this can lead to an odd number of capture moves, in which case one of the players has one capture move more than the opponent. A simple example is given in Figure 1.

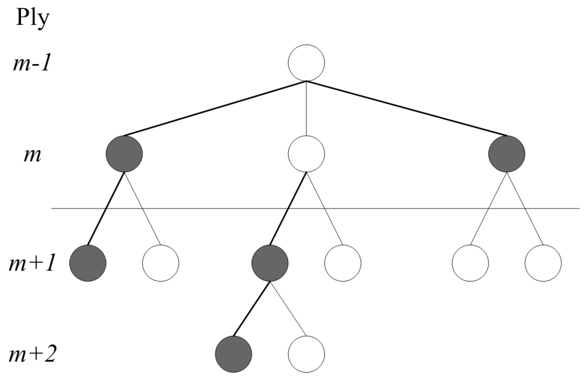

Figure 1: Expanded nodes with Q_2-limited search in an m-ply search. The grey nodes represent states following capture moves.

During an alpha-beta search, it often happens that on different locations in the search tree, identical situations occur. This happens if the same state can be reached via multiple sequences of moves. In order to take care of this, a transposition table [15] is used. An entry in the transposition table contains the hash key, the value of the corresponding node, a flag indicating whether the stored value is a bound or an exact value, the best move and the search depth. Transposion tables can also be used for move ordering.

Another way to order moves is by using killer moves. The killer-moves heuristic uses the assumption that a certain best move is not only the best one in the current situation, but also in similar ones. For each ply in the search tree, a small number of killer moves is stored. A move is a killer move if it has produced a cut-off in the search tree or if it turns out to be the best one. Since only a small number of killer moves is stored, the algorithm needs to make a selection. When a new killer move is found, the ‘oldest’ move in the list is replaced. At each node the killer moves of the corresponding ply are checked first. It is, however, possible that a killer move is not a valid move. Therefore, before playing a killer move, it first needs to be checked whether it is a valid move.

Another improvement we implemented is aspiration search. Instead of starting with search window $(-\infty, \infty)$, the search starts with $(V - \Delta, V + \Delta)$, where V is the expected value of the search tree and Δ is a small value. If the value of the game tree indeed lies between $V - \Delta$ and $V + \Delta$, then the value can be found much faster, since a smaller window may cause more cut-offs. If the value of the search tree lies outside the window, the search must be repeated using a larger window, i.e., $(-\infty, V - \Delta)$ if the search fails low and $(V + \Delta, \infty)$ if the search fails high. This means that it is important to choose a value for V such that it is close to the actual result. When using iterative deepening, the result of the previous iteration can be used as the value of V for the next one.

The last improvement is avoiding self-destruction. In Khet it is possible for a player to destroy one of his own pieces. During the alpha-beta search, self-destruction moves occur regularly. Even though in practice such moves will rarely be performed, they are still fully investigated. By avoiding self-destruction, all moves where a player captures one of his own pieces are ignored, since it can be assumed that such moves are rarely part of the perfect-play strategy of the player. As a result, large subtrees can be pruned, so the number of nodes that have to be investigated is decreased.

3.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [4] is a technique that can be used to play a large variety of games [2]. There are several reasons for choosing MCTS to play Khet. The first reason is that MCTS does not necessarily need any heuristic knowledge about the game. The only domain knowledge it needs are the rules and those are well defined. Contrary to alpha-beta search, it does not need any strategic knowledge. Since we do not have much strategic knowledge about Khet at hand, this can be an advantage for MCTS. The second reason is that a search tree of Khet has a large branching factor, at the start of the game around 80. Alpha-beta search can have some problems with trees with such high branching factors, while MCTS has proven itself with games with a branching factor much higher than Khet's, for example *Go* [3] and *Amazons* [10].

The basic workings of the Monte-Carlo algorithm are quite simple. Starting with the current board position, the game is finished by selecting random moves for both players. The simulation of one such game is called a sample. For selecting a move, a large number of samples is carried out. After the simulations are done, the algorithm chooses the move which has the highest win rate.

The disadvantage of the bare Monte-Carlo algorithm is that bad, unlikely moves are played about as often as good, likely ones. Often, a search tree is constructed that is used for storing statistics about the board positions that are represented by the nodes. These statistics can then be used for improving the search. There exist multiple Monte-Carlo Tree Search algorithms. The one that we investigate is called Upper Confidence bounds applied to Trees (UCT) [9], a best-first search algorithm, which allows the algorithm to play better moves more often than bad ones. This technique was succesfully used in the Go-playing-program MoGo, which has won multiple large Computer Go tournaments [14]. At each node of the tree, two values are stored: the number of times the node has been visited, and the number of times that passing this node resulted in a win. While traversing the search tree, at each node the UCT values of all children are calculated. The UCT values are calculated using the following formula [5]:

$$v_{UCT}(n,c) = \frac{w_c}{p_c} + C \times \sqrt{\frac{\ln(p_n)}{p_c}}$$

Here, n is the current node and c is the child. w_x represents the number of wins for node x and p_x the number of times node x has been visited. C is a constant value, which determines the exploration/exploitation trade-off. The best value for C should be determined experimentally. After calculating the UCT value of all children, the child with the highest UCT value is chosen. If at a node some children have not been investigated yet, which means a UCT value cannot be determined, one of these children is chosen, and a node representing this child is added to the tree. From this point on, the game is finished completely randomly, without adding any new nodes to the tree. When the game is finished, backtracking is used to store the result in the search tree.

Similar to an alpha-beta search tree, in a Monte-Carlo search tree transpositions can occur. A transposition table in a Monte-Carlo search tree works in a similar way as a transposition table in an alpha-beta search tree. In each entry, the number of visits of the node, the number of eventual wins, the hash key, and the depth are stored. If a transposition is found, then the statistics that are stored in the table can be used to calculate the UCT value of the child.

Games of Khet can take a long time to finish. Random games can take more than 1000 moves before they end. Such samples take a relatively long time to finish. In order to prevent such long samples, a maximum game length can be used. If after a certain number of moves, called d_{max}, the game is still not finished, the simulation is terminated and a heuristic value is returned. There are several ways to determine this heuristic value. The easiest way is to always declare the game a draw. This is a really fast heuristic, as no calculations are necessary.

4 Experiments and Results

In this section, we give a description of the experiments and their results. Some experiments are performed to determine the best parameter settings for some of the improvements. Also, the quality of all improvements is investigated. For all experiments we let two players with different settings play against each other. Both players receive 300 seconds per game. There is no maximum time per move. For Khet, there exist several standard set-ups [11]. In our experiments, all games start with the Original set-up.

4.1 Alpha-beta search

First, we investigate the performance of the improvements for the alpha-beta search algorithm. Before these experiments can be performed, we need to find the best parameter settings for some of the improvements.

4.1.1 Parameter settings

For Q_n-limited search, we need to perform experiments in order to determine the best value for n. n should not be too large, otherwise the algorithm will still search too deep. Also, n should not be too small, since otherwise the improvement may not have any effect. We let an alpha-beta player with Q_n-limited search play against an alpha-beta player without improvements, with various values for n. The results show that Q_n-limited search works best with $n = 2$, though it also gives good results with $n = 3$.

For aspiration search, we need to find the best value for Δ. Aspiration search works best if Δ is chosen such that the search window is narrow enough to produce extra cut-offs, but wide enough to still find the right value often enough. For these experiments, we enable Q_n-limited search with $n = 2$ for both players. Because of the horizon effect, the resulting value can change considerably between iterations, making it almost impossible to provide a reasonable expectation of the value of the next iteration. By enabling Q_n-limited search, these fluctuations are decreased. From the experimental results, we can conclude that aspiration search works best with $\Delta = 500$.

4.1.2 Results and discussion

In Table 2, the results for the various improvements for the alpha-beta search algorithm are summarised. For all sets of experiments, except the first two, Q_n-limited search with $n = 2$ is enabled for both players.

	As Silver			As Red			Total score
Improvement	wins	draws	losses	wins	draws	losses	(out of 50)
Quiescence search	4	0	21	5	0	20	9
Q_n-limited search ($n = 2$)	20	2	3	14	0	11	35
Transposition table	17	1	7	14	0	11	$31\frac{1}{2}$
Killer moves	20	2	3	21	0	4	42
Aspiration search ($\Delta = 500$)	14	2	9	13	1	11	$28\frac{1}{2}$
Incremental move generation	13	0	12	14	0	11	27
Avoiding self-destruction	13	2	10	10	2	13	25

Table 2: Experimental results for the alpha-beta algorithm with single improvements versus bare alpha-beta search.

It becomes immediately clear that the basic quiescence-search algorithm does not work well with Khet. There are several reasons for this. The first problem is that determining quiet moves in Khet is not a trivial task. When generating the list of valid moves, it is impossible to determine which moves will be quiet and which will not. In order to determine whether a move is quiet, the move needs to be performed, the laser needs to be fired and then it can be determined whether or not a piece is captured. This causes a large amount of overhead. Another problem is that in some occasions, almost all of a player's moves are capture moves. Situations like these occur regularly and cause the average branching factor during quiescence search to be relatively high. The third problem is that it is possible in Khet to have a large sequence of capture moves. Observation has shown that when applying quiescence search to a 2-ply deep search from the Original set-up, subtrees are built that reach depths of 16 ply or more.

The other results show that Q_n-limited search, transposition tables and killer moves turn out to be the best improvements. The win rates of incremental move generation, aspiration search and avoiding self-destruction are less than 60%. From these results we can conclude that these improvements do not cause the alpha-beta player to play much stronger, if any. It turns out, with aspiration search, that the first search fails too often. Both incremental move generation and avoiding self-destruction cause the player to search barely any deeper than without these improvements.

4.2 Monte-Carlo Tree Search

In this section we describe the experiments and results for the Monte-Carlo Tree Search algorithm. We investigate the strength of UCT and whether or not setting a maximum game length causes the MCTS player to play stronger. A transposition table is used if and only if UCT is enabled. The experiments are similar to those for the alpha-beta player. We let an MCTS player with one improvement play against an MCTS player without improvements. The settings are the same: again both players receive 300 seconds per game and all games start with the Original set-up.

4.2.1 Parameter settings

First, we need to find the best value for C in the UCT formula. If C is too high, then the algorithm will explore too much without exploiting the previous results. If C is too small, then the algorithm will start exploiting the previous results too early, possibly discarding good moves too fast. We let an MCTS player with UCT play against a bare Monte-Carlo player, with different values for C. From the results, we can conclude that UCT works best with $C = 0.375$. However, the best value of C is dependent on the number of games that can be simulated. With less games, C should be lower in order to exploit the previous results earlier, while if more games can be simulated, C should be higher as there is more time for exploration.

In order to prevent the Monte-Carlo algorithm from playing endlessly long games, we define a maximum game length. In order for this improvement to work as good as possible, we need to find the best value for d_{max}. If the value of d_{max} is too low, too few games are completed to give good results. If the value is too high, then the effect of this improvement decreases, since less games are cut off early. Experiments with varying values for d_{max} show that the MCTS player with $d_{max} = 100$ yields the best results.

4.2.2 Results and discussion

In Table 3 the results for UCT and maximum game length are summarised.

	As Silver			As Red			Total score
Improvement	wins	draws	losses	wins	draws	losses	(out of 50)
UCT ($C = 0.375$)	24	0	1	25	0	0	49
Maximum game length ($d_{max} = 100$)	17	0	8	19	0	6	36

Table 3: Experimental results for the MCTS algorithm with single improvements versus a bare Monte-Carlo algorithm.

For MCTS, it turns out that UCT is an indispensable improvement. Also, setting a maximum game length improves the performance of the MCTS player considerably, even without incorporating any domain knowledge into the evaluation function. Strategic knowledge can be implemented into the evaluation function, but this will make the evaluation considerably slower. Future research will be necessary in order to determine how strategic knowledge can be used in the evaluation function and whether it yields an increase of the performance, or whether the fact that less games are simulated decreases the strength of the player.

4.3 Alpha-beta search versus MCTS

Finally, we let the best alpha-beta player play against the best MCTS player in order to determine which algorithm works best. We let them play three tournaments, where in each tournament the players receive a different amount of time per game (60, 300 and 1800 seconds, respectively) and each tournament consists of a different amount of games (100, 50 and 10 games, respectively). For the alpha-beta player, we use a combination of a transposition table, killer moves and Q_n-limited search with $n = 2$. The MCTS player uses UCT with $C = 0.375$ and a maximum game length $d_{max} = 100$.

From the results, it becomes immediately clear that alpha-beta search works much better than MCTS. The alpha-beta player wins all games. Apparently, even the small amount of extra domain knowledge makes the alpha-beta player play on a much higher level than the MCTS player.

5 Conclusions and Future Research

In this paper, we investigated two different search techniques, both enhanced by various improvements, for playing the game Khet: alpha-beta search and Monte-Carlo Tree Search. Alpha-beta search can best be improved by implementing transposition tables, killer moves and the newly invented Q_n-limited search, a variation on quiescence search. MCTS can be enhanced by using UCT and by setting a maximum game length for the simulations.

The experimental results show that alpha-beta search works much better than MCTS. Even the small amount of domain knowledge that is incorporated into the evaluation function of the alpha-beta algorithm can make a great difference. Based on the analysis of the games, we can conclude that the MCTS player plays far too opportunistic. It also often captures one of its own pieces. The alpha-beta player can easily defend itself against the opportunistic play of the MCTS player, and always tries to defend its own pieces, while trying to attack the opponent's pieces.

Unfortunately, we were not able to test our program against experienced Khet players, but after playing a small number of games against the two players, we can conclude that the alpha-beta player is able to play Khet at a strong amateur level. The MCTS player, however, can only challenge a beginning player.

The MCTS algorithm can still be improved by implementing some strategic knowledge, but since the difference with the alpha-beta algorithm is so large, it is questionable whether MCTS will ever match the performance of alpha-beta search. This is a topic for future research.

The alpha-beta player can still be enhanced by, e.g., improving the evaluation function by incorporating more strategic knowledge, or by creating an opening book, which will cause the player to play much faster in the early stage of the game, causing it to have more time left during the mid-game and the endgame.

References

[1] D. F. Beal. A Generalized Quiescence Search Algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[2] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In M. Mateas and C. Darken, editors, *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217. AAAI Press, Menlo Park, CA., 2008.

[3] G. Chaslot, J.-T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik. Monte-Carlo Strategies for Computer Go. In P.-Y. Schobbens, W. Vanhoof, and G. Schwanen, editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 83–90. University of Namur, 2006.

[4] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630/2007 of *Lecture Notes in Computer Science*, pages 72–83, Turin, Italy, June 2006. Springer.

[5] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, INRIA, France, 2006.

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

[7] F.-H. Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NY, USA, 2002.

[8] D. E. Knuth and R. W. Moore. An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, 6:293–326, 1975.

[9] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006, Lecture Notes in Artificial Intelligence 4212*, pages 282–293. Springer, 2006.

[10] R. J. Lorentz. Amazons Discover Monte-Carlo. In H. J. van den Herik, Xinhe X., Zongmin M., and M. H. M. Winands, editors, *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] J. A. M. Nijssen. *Using Intelligent Search Techniques to Play the Game Khet*. M.Sc. Thesis, Maastricht University, Maastricht, The Netherlands, 2009.

[12] C. E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41:256–275, 1950.

[13] A. M. Turing. Digital Computers Applied to Games. In B. V. Bowden, editor, *Faster than Thought*, pages 286–310. Sir Isaac Pitman, London, 1953.

[14] N. Wedd. Computer go – past events. Online; `http://www.computer-go.info/events`.

[15] A. L. Zobrist. A Hashing Method with Applications for Game Playing. *Technical Report 88*, 1970. Reprinted (1990) in *ICCA Journal*, 13(2):69-73.