USING INTELLIGENT SEARCH TECHNIQUES TO PLAY THE GAME KHET

J.A.M. Nijssen

Master Thesis DKE 09-01

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of the Maastricht University

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
January 2009

Preface

Thank you for reading my master thesis. The research for this thesis was performed at the Department of Knowledge Engineering at the Maastricht University. The subject of the research is the implementation of Artificial Intelligence for a relatively new game called Khet, which is a two-player zero-sum game with complete information. This game is too complex to solve by present means, so we have to use Intelligent Search Techniques to create a computer program to play this game.

Of course, I am not the only person who made this thesis possible. Therefore, there are some people I want to thank.

First, and foremost, I would like to thank my supervisor, dr. ir. Jos Uiterwijk, for reading through and commenting on the progress of the thesis every week, for the brainstorm sessions during the research phase (which would eventually lead to the invention and implementation of incremental move generation and Q_n-limited search), and for his lectures during the course *Intelligent Search Techniques*. I also want to thank prof. dr. Jaap van den Herik, dr. Mark Winands and Jahn-Takeshi Saito, M.Sc. for their lectures during this very useful course.

Finally, I would like to thank my cousin, Yorick Wolswijk, and my uncle, Jos Wolswijk, for introducing this fascinating game to me. If it was not for them, I would never have thought of doing this research and investigating this game.

Pim Nijssen
Maastricht, January 2009

Abstract

For over 50 years, board games have been an important subject in the field of Artificial Intelligence (AI). Making an AI player that can beat the human world champion, and eventually solving the game, is the ultimate achievement for computer scientists. Over the course of these years, thousands of computer programs have been created to play hundreds of different games, some of them being able to even beat the best players in the world. Chess program Deep Blue beating Kasparov in 1997 is the most famous example.

Some games, like chess, have been investigated thoroughly by a lot of people for a long time. However, there also still exists a number of games that have not been investigated yet. One of these games is Khet, formerly known as Deflexion. Khet is a board game, played by two players, where strategic thinking is the key to success. It is a zero-sum game with perfect information.

This thesis describes the analysis of the game Khet and the implementation of the game engine and the AI players. First, the computations of the state-space complexity and the game-tree complexity are given. They turn out to be of the same order as those for 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 cause 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 reasonable level.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, first an introduction to games and computer games is given. Next, the problem statement and the research questions are stated. Finally, an outline of the thesis is described.

1.1 Games

Board games have been around as long as civilisation has. For thousands of years already, people have developed and played board games to challenge their intellect.

The first board games were invented over 5000 years ago. Around 3500 BC, the ancient Egyptians played a game called *Senet*. It was played on a board with 30 squares, where both players have to strategically move their team of draughtsmen over these squares. The exact rules of this game have long been forgotten, but it is known that this game was played for around 3000 years [20].

Also in China, people have been playing board games for thousands of years. The well-known board game *Go* originates in China and has been played there for around 4000 years. The game used to be most popular in East Asia, but in the recent years, the game has also become more popular in the rest of the world.

Over the past hundreds of years, the number of different games has grown enormously. Nowadays, games exist in almost every form or shape imaginable. They can be categorised by looking at properties that games have in common, for instance the number of players, the amount of information that is available, the presence of a chance element, and components that are required to play (for example a board and pieces, or playing cards).

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 ultimate challenge would be to make a computer player that is better than any human in the world.

1.2 Computer Games

The idea of having computers playing games has been around for over 50 years already. Shannon [24] and Turing [25] were the first ones in the early 1950s 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 [13].

During these years, many advancements have been made. Computers are getting faster every year and new techniques allow computers to work more efficiently or make them smarter. There are several reasons why games are an interesting subject for AI research.

First, games have well-defined rules. They are much more structured than real-life situations. This makes it easier to translate them to computer programs [11]. But even though the rules are often pretty simple, playing the game well is difficult. Most games are easy to learn, but hard to master [17].

Games are well suited for testing new problem-solving techniques. If it turns out that these techniques work well for playing games with a computer, they can often also be used in other research areas [19].

Another reason why games are an interesting research subject is that by creating a computer program that is able to play games, researchers may get more insight in the way humans think and reason [10, 18].

Chess has always been the most important game in the research field of Artificial Intelligence, but recently a lot of other games have also gained a lot of popularity. This has lead to the solving of various games, like *Nine Men's Morris* [7], *Connect Four* [1] and *Awari* [21]. Some other games, like *Othello*, can already be played on a super-human level, meaning that computer programs are already able to beat the best players in the world. There are, however, also still quite some games where humans are still better than computer players, like *Go*.

1.3 Problem Statement and Research Questions

Nowadays, still a lot of new board games are invented. One game that combines the simplicity and abstract of classic board games with modern techniques is called *Khet*. Khet was invented in 2004 under the name *Deflexion*. It can be seen 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 before.

The goal of this research is to make a computer program that is able to play Khet as good as possible. Thus, the problem statement is defined as follows:

> *Is it possible to develop a computer program to play the game Khet effectively and efficiently?*

In order to give an answer to this problem statement, we define some research questions in order to break the problem into smaller sub-problems. First, it is important to know how complex the game is:

What is the game complexity of Khet?

After answering this question, some search techniques can be selected to let the program play Khet:

Which search techniques can be used to play Khet?

These search techniques can then be refined in order to make them work more efficiently, so that they can perform a more extensive and/or more efficient search in the same amount of time:

How can these search techniques be improved to increase the strength?

Finally, after answering the previous research questions, experiments can be performed in order to answer the final question:

Which combination of techniques has the best balance between speed and strength?

1.4 Outline of the Thesis

The thesis is structured as follows:

- Chapter 1 gives a general introduction to games and computer games and describes the problem statement and the research questions.
- Chapter 2 gives an introduction to Khet. It describes the rules of the game and some strategies.
- Chapter 3 describes the complexity analysis of Khet. This complexity analysis consists of the analysis of both the state-space complexity and the game-tree complexity.
- Chapter 4 gives a description of the search techniques that have been used, including the improvements that have been implemented. Also, some important implementational details are given.
- Chapter 5 gives a description of the experiments that have been performed and shows the results.
- Chapter 6 lists the conclusions that can be drawn from the experimental results and answers the research questions and, consequently, the problem statement.

Chapter 2

The Game Khet

Khet [14] 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.

In this chapter, a description of the game Khet is given, including the rules and some common strategies.

2.1 The 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.

Each player receives 14 pieces at the start of the game: 7 pyramids, 4 obelisks, 2 djeds and 1 pharaoh (see Section 2.3). Some of these pieces have mirrors attached to them. 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 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. There are exceptions to this rule, which will be explained in Section 2.3.

Then, the player has to fire his laser, which is mounted inside the edge of the board. If this laser hits a piece on a mirrored side, the laser beam 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 both pharaohs is hit by a laser. 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 next player can claim a draw. Two board positions are the same when the same type of pieces with the same colors occupy the same squares with the same orientation.

2.2 The Board

The board consists of 80 squares: 60 grey (neutral) squares, 10 red squares and 10 silver squares. On silver squares, only silver pieces are allowed, while on red squares, only red pieces are allowed.

There are two lasers mounted inside the edge of the board. The laser of Silver is located at the bottom of column j and the laser of Red is located at the top of column a. This is illustrated in Figure 2.1.

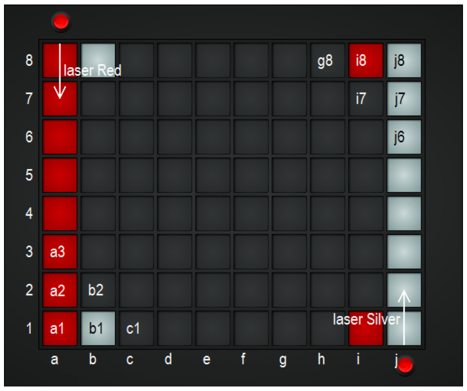


Figure 2.1: Khet board.

2.2.1 Notation

There exists no official notation for the board locations and the moves in Khet, so we will introduce an unofficial notation that will be used in this thesis.

For the indication of a board location, a notation is used that is similar to chess. The columns are denoted by a letter, 'a' to 'j' from left to right, and the rows by a number, '1' to '8' from bottom to top. In the notation, first the column letter is given and then the row number. This means that the bottom left corner of the board is denoted by *a1* and the upper right corner by *j8*.

When moving a piece, the location of the piece to move and the target location are written down. For instance, moving the pyramid from *j4* to *j3* is denoted as *j4j3*. For denoting the rotation of a piece, the location of the piece to rotate is written, followed by either *r+*, indicating a clockwise rotation, or *r-*, indicating a counter-clockwise rotation. So, rotating the pyramid on *h2* clockwise is denoted as *h2r+*. For djeds, it does not matter in which direction you rotate them, since the result is in both cases the same. In that case, the + or - can be omitted. Rotating the djed on *f4* can be denoted as *f4r*.

When moving a stacked obelisk (see Section 2.3.3), the standard notation is used. However, if only one of the two obelisks is moved, so the stacked obelisk is unstacked, then the letter *u* is added between the piece location and the target location. For example, unstacking the stacked obelisk at *d1* by moving one of the obelisks to *e2* is denoted as *d1ue2*.

If a piece is captured, the move notation will be appended by an *x* and the location of the captured piece. For example, if the piece at *g3* is captured after the move *j4j3*, this is written down as *j4j3xg3*.

2.2.2 Set-ups

Contrary to, for example, chess, Khet starts with the pieces of both players spread over the board. Also, there is no officially fixed starting configuration; players can use their own imagination to create new starting positions. The makers of Khet have defined three possible starting configurations that give a good balance between offense and defense and gives neither player a considerable advantage:

- Original (see Figure 2.2)
- Imhotep (see Figure 2.3)
- Dynasty (see Figure 2.4)

2.3 The Pieces

Khet contains four different types of pieces: djeds, pyramids, obelisks, and pharaos. Table 2.1 shows more information about these different types.

2.3.1 Djeds

Djeds are the most powerful pieces in the game. They have two mirrors, so they cannot be captured by a laser. Djeds have the ability to swap places with (stacked) obelisks and pyramids. This is only allowed if the two pieces are adjacent to each other. Djeds can not only swap with pieces of the same color, but also with pieces of the opposing color. This is only allowed if neither piece lands on a square of the opposing color. So a silver djed may not swap with a piece on a red square. Also, a djed on a silver square may not swap with a red piece, since then the red piece would land on a silver square, which is not allowed.

Since djeds have no unprotected sides, they can never be removed from the board. Because of this property, they are ideally suited for attacking purposes. They can also be used as defenders, but pyramids can do this task just as well.

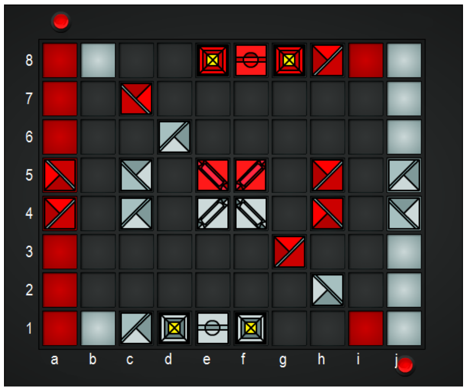


Figure 2.2: Original Setup.

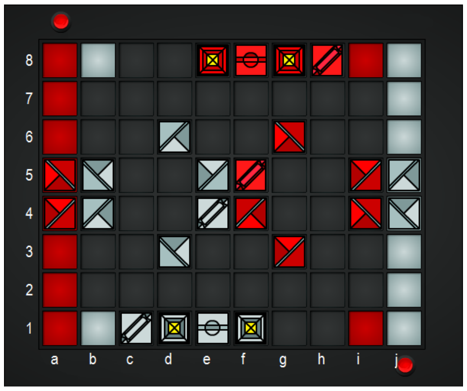


Figure 2.3: Imhotep Setup.

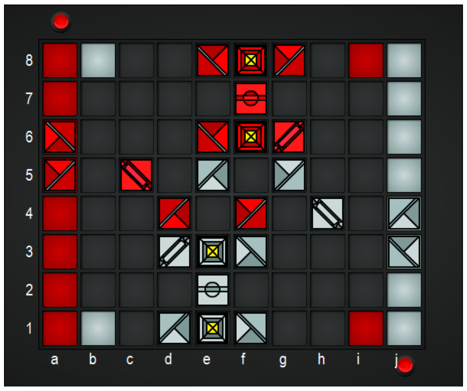


Figure 2.4: Dynasty Setup.

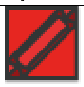
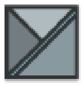
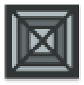
Name	Symbol	Mirrors
Djed		2
Pyramid		1
Obelisk		0
Stacked obelisk		0
Pharaoh		0

Table 2.1: Khet Pieces.

2.3.2 Pyramids

Pyramids are moderately powerful pieces. They have one mirror and two unprotected sides. They can be used to attack the opponent, but they have to be used with care, since they can also be attacked by the opponent. Pyramids are also often used to defend the pharaoh.

This makes pyramids the most balanced pieces of the game. Also, they have no special abilities.

2.3.3 Obelisks

Obelisks are the weakest pieces in the game, since they do not have mirrors. They do, however, have a special ability. Obelisks can be stacked on top of each other, making it possible to move two obelisks at once. When making a move, a player is allowed to stack two obelisks, as long as they belong to the same player and if they are adjacent to each other. Also, a stack may not be larger than two obelisks. Making triple or quadruple stacks is not allowed. Whenever a stack of obelisks is hit by a laser, only the top obelisk is removed.

Obelisks may also be unstacked. When unstacking, the player places the top obelisk on an empty square adjacent to the location of the stacked obelisk. Both stacking and unstacking count as one move.

Since obelisks do not have any mirrors, they can only be used as defenders. The main function of obelisks in Khet is to protect the pharaoh from the lasers.

2.3.4 Pharaohs

The pharaohs are the most important pieces in the game. If a player's pharaoh is hit by a laser, he loses the game. A pharaoh has no mirrors, so it cannot defend itself.

2.4 Strategies

Khet is still a new game. It is also a rather unknown one. This means that it is it difficult to find strategies that can be used as domain knowledge in the search algorithms. However, we can define some basic strategies that are useful to make a computer program play Khet at a reasonably high level.

2.4.1 Protecting

Whenever one's pharaoh is hit by a laser, he loses. Since a pharaoh cannot protect itself with a mirror, one needs to protect his pharaoh with other pieces. Since obelisks do not have mirrors, they can only fulfill defensive tasks. Usually, both players start with their obelisks in two stacks of two pieces, adjacent to their pharaohs. These stacks of obelisks can be used to block a laser beam from the opponent, in order to prevent the pharaoh from being hit. This is done by moving a stack of obelisks in the opponent's laser route. Now the opponent will have to take two turns to capture the obelisks, giving the player two turns to set up his defenses, instead of one. So keeping the obelisks close to the pharaoh is an important strategy.

Usually, both players start with two stacks of two obelisks. It is often not a good idea to unstack these obelisks. If they are stacked, the player can move both of them in one turn, increasing the mobility of his pieces. Unstacked obelisks are also pretty useless. If a player uses a single obelisk to prevent the opponent from hitting the pharaoh with his laser beam, the opponent will hit the obelisk, leaving the pharaoh just as vulnerable as before, while he has lost one piece. If the player uses a stack of two obelisks to protect his pharaoh, he has one extra turn before the pharaoh will be exposed. This gives him time to organise his defenses. So generally speaking, to unstack obelisks is not a good move.

2.4.2 Attacking

Besides defending his own pharaoh, a player should also try to attack his opponent's pharaoh. Attacking is considerably more difficult than defending. Also, since Khet is a relatively unknown game, there are not many strategies available. There are, however, some basic rules that every player applies.

With the standard set-ups, both players start with two pyramids on their own column. One of these pyramids is used to reflect the laser beam into the playing field. This pyramid should not be rotated or moved from its column. Otherwise, the player's laser beam is not deflected from his own column, making it impossible to attack the opponent. Theoretically, also a djed can be used for this purpose, but this is useless. Djeds are much more useful when they are used inside the playing field. Moreover, one of its mirrors will not be used if it is positioned on the player's own column.

Chapter 3

Complexity Analysis

In order to get some insight into the game Khet, the complexity of the game has to be determined. There are two types of complexity measures: state-space complexity and game-tree complexity. In this chapter, both of these complexities of Khet are calculated. At the end of this chapter, the results are used to compare Khet to a number of other board games.

3.1 State-Space Complexity

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 way too large.

Determining the state-space complexity in Khet is rather tricky. There are several rules which make it very difficult to give an exact calculation of the number of possible game states in Khet. Therefore, only a rough estimate is given. A Khet board consists of 80 squares: 10 silver ones, 10 red ones and 60 neutral ones. First, let us consider only the 14 silver pieces. These 14 pieces can be placed on 70 different squares. This leads to the following amount of possible ways to place the pieces:

$$n_{0,silver} = \binom{70}{1} \cdot \binom{69}{7} \cdot \binom{62}{2} \cdot \binom{60}{4} = 6.9641 \cdot 10^{19}$$

Here, the first term accounts for the placement of the pharaoh on 70 possible locations, the second for the placement of the 7 pyramids on the remaining 69 squares, the third for the placement of the 2 djeds and the fourth for the 4 obelisks.

This, however, does not take into account the possible rotations of the pieces. A pyramid can have four different orientations, while a djed can have two different ones. Theoretically, obelisks and pharaohs can also be rotated, but their orientation does not affect the game state in any way. This leads to the following, updated formula:

$$n_{0,silver} = \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{4} = 4.5640 \cdot 10^{24}$$

After the silver pieces have been placed on the board, we consider the red pieces. The problem is that there is no fixed amount of possible squares where the red pieces can be placed. Red pieces can only be placed on the 10 red squares and the remaining neutral squares, but it depends on the setup of the silver pieces how many neutral squares are left. Of course, it is possible to make an exact computation, but this is painstakingly difficult and the result will be unnecessarily accurate. So the assumption is made that, on average, Silver places 2 of his pieces on a silver square and 12 of his pieces on neutral ones. This is a reasonable assumption because 1 out of 7 squares where Silver can place his pieces is silver, so on average 1 out of 7 pieces will be placed on a silver square. This leaves Red with 58 squares (10 red + 48 neutral) to place his pieces. This leads to the following formula, which is roughly the number of ways to place all 28 pieces on the board:

$$n_0 = \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{4} \times \binom{58}{1} \cdot 4^7 \binom{57}{7} \cdot 2^2 \binom{50}{2} \cdot \binom{48}{4} = 1.0933 \cdot 10^{48}$$

The stacking rule in Khet allows two obelisks of the same player to be stacked on top of each other. However, in the previous calculation it is assumed that obelisks cannot be stacked.

Table 3.1 shows in which ways the obelisks can be stacked. Table 3.2 gives another representation, which can be used to simplify the calculations. This leads to the following calculation:

$$\begin{aligned}
n_0 \quad = \quad & 1 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{4} \cdot \binom{56}{0} \times \binom{58}{1} \cdot 4^7 \binom{57}{7} \cdot 2^2 \binom{50}{2} \cdot \binom{48}{4} \cdot \binom{44}{0} + \\
& 2 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{4} \cdot \binom{56}{0} \times \binom{58}{1} \cdot 4^7 \binom{57}{7} \cdot 2^2 \binom{50}{2} \cdot \binom{48}{2} \cdot \binom{46}{1} + \\
& 2 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{4} \cdot \binom{56}{0} \times \binom{58}{1} \cdot 4^7 \binom{57}{7} \cdot 2^2 \binom{50}{2} \cdot \binom{48}{0} \cdot \binom{48}{2} + \\
& 1 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{2} \cdot \binom{58}{1} \times \binom{59}{1} \cdot 4^7 \binom{58}{7} \cdot 2^2 \binom{51}{2} \cdot \binom{49}{2} \cdot \binom{47}{1} + \\
& 2 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{2} \cdot \binom{58}{1} \times \binom{59}{1} \cdot 4^7 \binom{58}{7} \cdot 2^2 \binom{51}{2} \cdot \binom{49}{0} \cdot \binom{49}{2} + \\
& 1 \cdot \binom{70}{1} \cdot 4^7 \binom{69}{7} \cdot 2^2 \binom{62}{2} \cdot \binom{60}{0} \cdot \binom{60}{2} \times \binom{60}{1} \cdot 4^7 \binom{59}{7} \cdot 2^2 \binom{52}{2} \cdot \binom{50}{0} \cdot \binom{50}{2} \\
= \quad & 1.7711 \cdot 10^{48}
\end{aligned}$$

This number does not include the positions where one or more pieces are removed from the board. In order to get an idea about how the number of possible positions decreases if a number of pieces is removed from the board, we will now calculate the number of possible locations with one piece removed from the board.

	Silver		Red	
Index	Unstacked	Stacked	Unstacked	Stacked
1	4	0	4	0
2	2	1	4	0
3	0	2	4	0
4	4	0	2	1
5	2	1	2	1
6	0	2	2	1
7	4	0	0	2
8	2	1	0	2
9	0	2	0	2

Table 3.1: Possible ways to stack obelisks.

	Player 1		Player 2		
Index	Unstacked	Stacked	Unstacked	Stacked	Occurrences
1	4	0	4	0	1 (1)
2	4	0	2	1	2 (2, 4)
3	4	0	0	2	2 (3, 7)
4	2	1	2	1	1 (5)
5	2	1	0	2	2 (6, 8)
6	0	2	0	2	1 (9)

Table 3.2: Possible ways to stack obelisks. The values between brackets under Occurences correspond to the indices of Table 3.1.

The only two pieces that can be removed from the board are pyramids and obelisks. If a pharaoh is hit, the game is over, so a position with only one pharaoh cannot occur during a game. Also, djeds cannot be removed from the board, so board positions where one of both players has only one djed are also illegal.

The following calculation shows the estimated number of positions where one piece is removed from the board:

$$\begin{aligned} n_1 &= 2 \cdot \binom{70}{1} \cdot 4^7 \cdot \binom{69}{7} \cdot 2^2 \cdot \binom{62}{2} \cdot \binom{60}{4} \times \binom{58}{1} \cdot 4^6 \cdot \binom{57}{6} \cdot 2^2 \cdot \binom{51}{2} \cdot \binom{49}{4} + \\ &\quad 2 \cdot \binom{70}{1} \cdot 4^7 \cdot \binom{69}{7} \cdot 2^2 \cdot \binom{62}{2} \cdot \binom{60}{4} \times \binom{58}{1} \cdot 4^7 \cdot \binom{57}{7} \cdot 2^2 \cdot \binom{50}{2} \cdot \binom{48}{3} \\ &= 2.7939 \cdot 10^{47} \end{aligned}$$

In this calculation, the first line represents the situation where a pyramid is missing, and the second where an obelisk is missing. Note that both calculations are multiplied by a factor 2. This is to accomodate for both situations where Silver or Red is the player who is missing one piece.

This calculation does not include stacked obelisks, but from the previous calculations we can conclude that they do not increase the number of positions significantly. This increase will even be relatively lower, since one of the obelisks can be removed from the board.

From this calculation we can conclude that if one piece is removed from the board, the number of possible positions drops significantly. From this we can assume that if one or more pieces are removed from the board, the number of possible positions is small compared to the number of positions when all pieces are present. This means that any further calculations are unnecessary and that the state-space complexity of Khet can be assumed to be 10^{48}, with an upper bound of 10^{48}.

3.2 Game-Tree Complexity

The game-tree complexity denotes how many different games can be played. In other words, it indicates how many terminal nodes a complete search tree has. The game-tree complexity can be computed using the formula $GTC = b^d$. 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 a player has and d is the average length of a game.

Khet is a relatively unknown game. 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.

Determining the branching factor is not very difficult. While simulating a game, each turn the number of valid moves is stored and finally, the average of these numbers is taken.

Figures 3.1, 3.2 and 3.3 show the frequencies of the branching factors in a large number of games for three different AI's. All graphs have a peak at a branching factor of 80. This is because at the start of the game the first player can make 80 different moves. The average branching factor for each type of AI is displayed in Table 3.3.

AI	Average branching factor
random	63.81
alpha-beta (depth 2)	67.10
alpha-beta (depth 4)	68.92

Table 3.3: Average branching factor for three different AI's (simplified).

Determining the average game length is considerably more difficult. The length of a game can vary drastically. Table 3.4 shows the average game length for the same players in the same games as previously described.

Contrary to the average branching factor, the average game length varies drastically for different AI's. These differences can be explained. The random AI plays completely randomly and, besides 'accidentally' hitting pieces of the opponent, might also hit his own pieces, including the pharaoh. The depth-2 alpha-beta AI is only concerned with trying to hit pieces of the opponent and

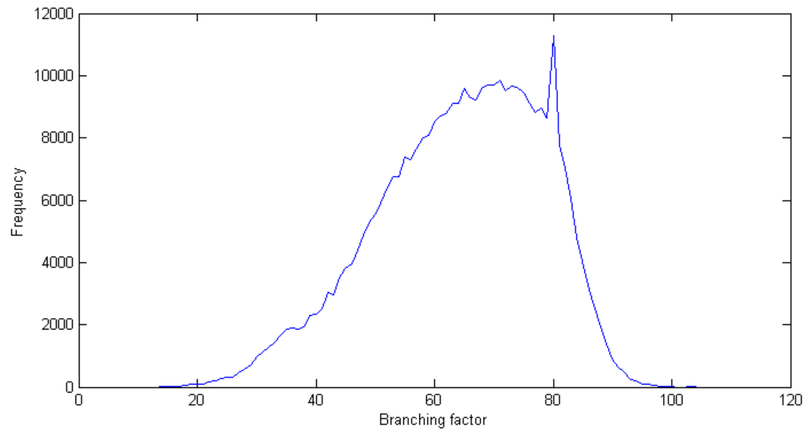


Figure 3.1: Frequency of branching factors in 1000 games played by two random AI's.

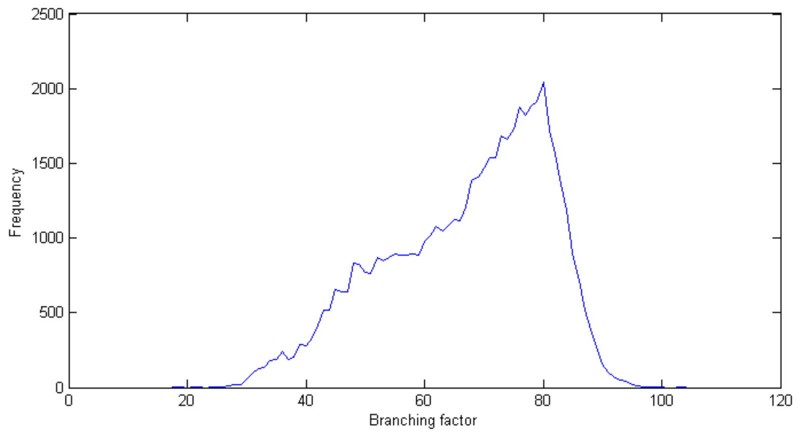


Figure 3.2: Frequency of branching factors in 100 games played by two alpha-beta AI's with a fixed depth of 2.

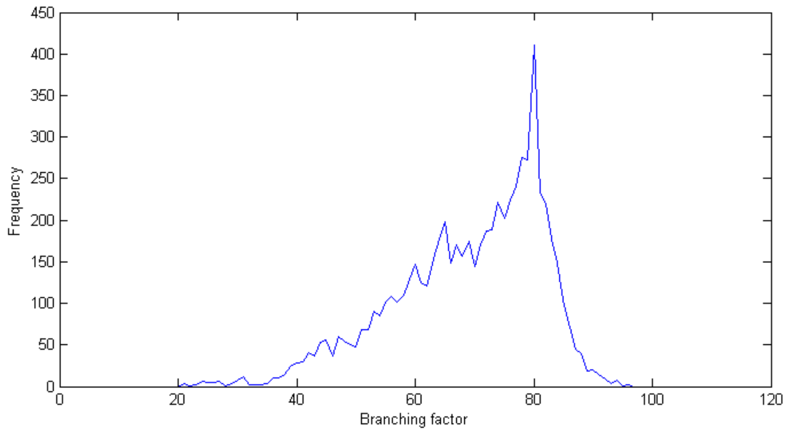


Figure 3.3: Frequency of branching factors in 100 games played by two alpha-beta AI's with a fixed depth of 4.

AI	Average game length
random	154.9
alpha-beta (depth 2)	557.6
alpha-beta (depth 4)	67.95

Table 3.4: Average game length for three different AI's.

preventing the opponent from hitting his own. This makes both AI's very protective and as a result a game can take hundreds, or sometimes even thousands of turns. The depth-4 alpha-beta AI looks further ahead and is willing to sacrifice one of his pieces to capture one of the opponent. This makes the games with these AI's considerably shorter.

Since the depth-4 alpha-beta AI is assumed to be the strongest player and gives the most plausible result, we use the average branching factor and the average game length of the alpha-beta player with depth 4, and neglect the results of the other two AI's.

From these experiments, we can conclude that the average branching factor in Khet is 69 and the average game length is 68. This leads to the following computation of the game-tree complexity: $GTC = 69^{68} \approx 10^{125}$.

3.3 Comparison with Other Games

To put the complexities in perspective, we need to compare them to those of other games. Table 3.5 shows the state-space complexity and the game-tree complexity for a number of games [12], including Khet.

From these numbers we can conclude that both the state-space complexity

Game	log state-space compl.	log game-tree compl.
Awari	12	32
Checkers	21	31
Chess	46	123
Chinese Chess	48	150
Connect-Four	14	21
Dakon-6	15	33
Domineering (8×8)	15	27
Draughts	30	54
Go(19×19)	172	360
Go-Moku (15×15)	105	70
Hex (11×11)	57	98
Kalah(6,4)	13	18
Khet	**48**	**125**
Nine Men's Morris	10	50
Othello	28	58
Pentominoes	12	18
Qubic	30	34
Renju (15×15)	105	70
Shogi	71	226

Table 3.5: State-space complexity and game-tree complexity for various games.

and the game-tree complexity for Khet are comparable to chess.

This means that Khet is, just like chess, 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.

Chapter 4

Implementation

In this chapter, 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. In Chapter 3, we saw that the complexity of Khet is close to the complexity of chess. Alpha-beta search has proven to be a good search technique for playing chess, so it is likely that alpha-beta search can also give good results for Khet. MCTS is a technique that is used not as often as alpha-beta search, but it has given very 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.

4.1 Alpha-Beta Search

The alpha-beta search technique 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.

The idea for the alpha-beta algorithm was proposed by John McCarthy at the Dartmouth Conference in 1956. Alexander Brudno, who published his results in 1963, was the first one to make a thorough investigation of the alpha-beta algorithm [4]. In 1975, Donald E. Knuth and Ronald W. Moore refined the algorithm and proved its correctness [15].

For a detailed explanation of the alpha-beta search algorithm, please refer to [22].

4.1.1 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.

As explained previously, since Khet is still a very new game, there is not much information about strategies available. The implemented evaluation function only looks at the most basic strategic elements.

The first and most important element is the number of pieces. Usually, having more pieces than your opponent is better than having less. 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.

Obelisks are the weakest type of pieces in the game. Therefore, they have the lowest value, in our case 10,000 points. Stacked obelisks are two obelisks stacked on top of each other. However, as explained in Chapter 2, stacked obelisks are more useful than two separate obelisks. Therefore, the value of a stacked obelsisk is more than twice as large as the value of a single obelisk. In our implementation, the value of a stacked obelisk is 25,000 points.

Since the main task of the obelisks is to protect the pharaoh, they become more useless as they move further away from the pharaoh. So the value of the (stacked) obelisks should decrease when they are further away from their pharaoh. To take care of this, the basic value (10,000 points for an obelisk, 25,000 points for a stacked obelisk) is divided by their Manhattan distance to the pharaoh. If the obelisks are adjacent to the pharaoh, the Manhattan distance is 1, so only then they receive their basic score. Because of this, obelisks will stay close, preferably adjacent, to their pharaoh.

Pyramids are moderately powerful pieces in the game and for each pyramid on the board, the player receives 75,000 points. The position or orientation of the pyramid does not affect the value of the piece, except for one. One pyramid should be used to send the player's laser beam into the playing field. If such a pyramid exists, a small bonus of 5000 points is awarded.

Djeds are the most powerful pieces of the game. However, djeds do not receive any points for the player. This is because djeds can never be destroyed and thus are present on every legal board position. They also do not receive any bonuses for their position or for sending the laser beam into the playing field. The latter is because a pyramid can do this job as well.

Finally, a random value is added to the score. This is a 12-bit value, so it lies between 0 and 4095. This random value is added to ensure that the computer player will play different moves in identical situations. If this random value would not be added, the computer player would be deterministic, so then it would be possible to beat it every time by playing the same sequence of moves.

4.1.2 Iterative deepening

Since alpha-beta search is a depth-first search algorithm, it always first expands the deepest unexpanded node.

As explained before, Khet is too complex to make a search through the whole tree. That is why a maximum depth of the search tree needs to be set. However, instead of setting a fixed maximum depth of the search tree, it is also possible to use iterative deepening.

Instead of performing one search up to a fixed depth, a sequence of tree searches is performed, where for each following search the maximum search depth is increased.

Intuitively, it might seem that iterative deepening causes a large waste of time. However, it turns out that this waste of time is relatively small. If we traverse a search tree of depth 4 with a branching factor of 80, then the number of investigated nodes will be (without pruning):

$$1 + 80 + 6,400 + 512,000 + 40,960,000 = 41,478,481$$

With iterative deepening, the number of investigated nodes will be

$$\begin{array}{llcrl}
\text{Search 1:} & 1 + 80 & = & 81 & \\
\text{Search 2:} & 1 + 80 + 6,400 & = & 6,481 & \\
\text{Search 3:} & 1 + 80 + 6,400 + 512,000 & = & 518,481 & \\
\text{Search 4:} & 1 + 80 + 6,400 + 512,000 + 40,960,000 & = & \underline{41,478,481} & + \\
 & & & 42,003,524 &
\end{array}$$

This shows that the total number of investigated nodes is only 1.27% higher than without iterative deepening. This percentage will be even lower with deeper searches.

Beside the fact that iterative deepening causes barely any overhead, there are also some advantages to using this technique.

The first advantage of iterative deepening is that the program has more control over the time that is spent on a certain move. This is especially useful in a timed game. Without iterative deepening it is quite hard to predict how long it will take to do a search up to a certain depth, because search times can vary significantly for different positions, even with similar search depths. Iterative deepening allows, using the spent times of the previous searches, to give a reasonable estimate of how long the next search will take. This information can then be used for determining whether the program should perform the next search or whether it should return the results of the last search.

Also, some of the data of the previous search can be used to improve the speed of the next search. There are several techniques for using this data, including transposition tables, killer moves and aspiration search. By using these techniques, the number of nodes that have to be investigated can be considerably reduced. So as a result, by using iterative deepening to some depth, often less nodes are investigated than when using a fixed-depth search to the same depth.

In the next sections, we will explain more about these techniques.

4.1.3 Incremental move generation

One way to speed up the alpha-beta algorithm is by implementing incremental move generation (IMG). Originally, 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. IMG 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.

4.1.4 Quiescence search

One large disadvantage of the alpha-beta search technique is the horizon effect. The search tree is traversed up to a fixed depth and then a heuristic is used to determine a value for the current position. However, it is possible that one move further the situation is entirely different. In order to take care of this, quiescence search [2] can be used.

Normally, whenever the search algorithm reaches a leaf node, the heuristic value of the node is returned. When quiescence search is applied, the algorithm searches for 'noisy' moves. These are moves that cause a large change in the evaluation of the board position. There are several ways the evaluation value of the board position can change, as was explained in Section 4.1.1, but capture moves are the most important ones.

So 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.

Contrary to most of the other improvements, quiescence search may cause more nodes to be investigated, instead of less. Because of this, the algorithm can often search one or two plies less deep than without quiescence search. However, the results for each ply are more reliable, so this offsets the disadvantage of searching less deep.

Q_n-limited search

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. In Chapter 5 we will show that it is possible in Khet to create very long sequences of capture moves, so 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.

Of course, 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.

Note that Q_∞-limited search works in the exact same way as the basic quiescence-search algorithm.

4.1.5 Transposition tables

During an alpha-beta search, it often happens that on different locations in the search tree, identical situations occur. For example, consider the position in Figure 4.1. This position can be reached from the Original setup (see Figure 2.2) via the moves *1. j4j3 h4r+ 2. h2h3*, but also via *1. h2h3 h4r+ 2. j4j3.* Even though both move sequences lead to the same position, this position occurs in two different locations in the search tree.

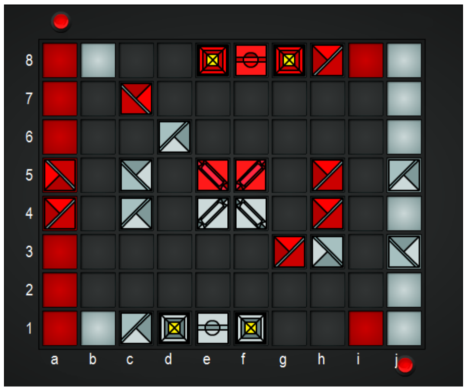


Figure 4.1: Early position which is reachable in two different ways.

Whenever the search algorithm arrives at a node which represents a situation that has been evaluated in the past, it is a waste of time to perform the evaluation of the node again. Instead, the results of the evaluation of the previous node should be used. In order to take care of this, a transposition table (first described in [9]) is used.

Zobrist hashing

Ideally, we would store every position that has been evaluated in the past, but unfortunately this is not possible. Computers nowadays have not enough memory to store all this data. Therefore, we use a transposition table including a hashing method.

A transposition table consists of 2^n entries, where each entry can hold information about a certain position. In order to determine where each position is stored, a hash function called Zobrist hashing is used [27].

First, for each possible piece with each possible orientation on each possible square, a random 64-bit number is generated. This means that in total 3200 numbers are generated, since there are 80 squares, 10 different types of pieces (five, including the stacked obelisk, for each player) and each piece can have a maximum of four different orientations. The Zobrist value is calculated by performing an XOR operation on the random numbers corresponding to all pieces, including their orientation and position, on the board. This leads to a 64-bit number which is the hash value of the current position.

Of this hash value, the last n bits are used to determine the location of the current board position in the transposition table. The other $64-n$ bits are used as the hash key, which will be explained next.

Table entries

An entry in the transposition table contains the following elements:

- The hash key
- The value of the corresponding node
- A flag indicating whether the stored value is a bound (in case of a cut-off) or a value
- The best move
- The search depth

Using the transposition table

At the start of every node evaluation, it is first checked whether the Zobrist value of the board corresponding to the node already exists in the transposition table. If it does, which means that also the hash key matches, then the information stored in the transposition table can be used. If the hash key does not match, then the information cannot be used, as the Zobrist value, and thus the position, is obviously different. If the previously searched depth of the node is at least the depth that currently has to be searched and the result was an exact value, then this value can immediately be returned as the value of the current node. If the result was a bound, then α is set to the value of the bound if this value is higher than the current value of α. If this makes α higher than the current value of β, the value of α can be returned. Otherwise, the search still has to be executed. The move that was considered to be the best one previously (i.e., the move which caused the cut-off) is during this search investigated first, as there is a good chance that this move will produce a cut-off again. If the previously searched depth of the node is smaller than the depth that currently has to be searched, most of the information cannot be used. The only information that can be used is the best move according to the transposition table. This best move will be investigated first, as this is the move which has the highest chance to produce a cut-off.

Errors

Using a transposition table is not without risks. There are chances that two different board positions need to be stored on the same entry in the table. When using transposition tables, there are two types of errors that can occur: type-1 errors and type-2 errors.

Type-1 errors occur when two different board positions have the exact same hash value. It is very difficult to recognise this type of error, as the hash key for both board positions is the same. One possible way to detect this type of error is checking whether the stored best move is a legal move. If it is not, we can be assured that we have found a type-1 error. If it is, the error will go unnoticed and might lead to wrong evaluations.

A type-2 error, also called a collision, occurs when two different board positions with two different hash values are assigned the same table entry. In other words, the last n bits of the hash values of the two board positions are the same.

This type of error is easily recognised, since the rest of the hash value is stored in the entry as a hash key. As explained before, if an entry is retrieved from the table with a different hash key than the hash key of the current board position, then the data cannot be used.

Whenever the evaluation of the board position is finished, either by having evaluated all children or by performing a cutoff, the node information has to be stored in the transposition table. A problem can be that the information of another node is already stored at the location where the information of the current node should be stored. This means that the data of one of the two nodes has to be omitted. A replacement scheme is used to determine which node should be stored. One of the most commonly used replacement schemes is called Deep [3]. When using this replacement scheme, the node with the largest search depth is stored in the table, since this node contains more information and thus is, in general, more useful. If both nodes have the same search depth, then the old one is replaced by the new one.

4.1.6 Killer moves

Alpha-beta search works best if good moves are investigated first. As explained above, a transposition table can be used to store the best move for each node. Another way to order moves is 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 can be 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, possibly after the best move stored in the transposition table. Contrary to the best moves stored in the transposition table, it is 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.

In our program, we store two killer moves per ply.

4.1.7 Aspiration search

Typically, the alpha-beta search starts with a search window between $-\infty$ and ∞. The speed of the search can be increased if a smaller window is used, because a smaller window will cause more cutoffs. The smaller the window, the faster the search will be. The technique of using smaller search windows is called 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 will be found much faster than when the search starts with the window $(-\infty, \infty)$.

There is, however, a risk that the value of the tree lies outside the specified window. This is the case when the result of the search is smaller than or equal to $V - \Delta$ or larger than or equal to $V + \Delta$. In the former case, we say that the search fails low. In this case, the search needs to be performed again, but now starting with search window $(-\infty, V - \Delta)$. In the latter case, the search fails high and a re-search will be performed, starting with the search window $(V + \Delta, \infty)$.

In order to make aspiration search as efficient as possible, it is critical to make a good estimation of the value of V. Often, when using iterative deepening, the result of the previous search can be used as an estimate for the next search. So at depth d, the value of the search tree with depth $d - 1$ can be used as the value of V.

Also, the choice of the value of Δ is important for the efficiency of the technique. This value should not be too small, because otherwise the first search will fail too often. It also should not be too large, since otherwise there will be no gain compared to starting with the maximum search window. The best value of Δ is very dependent on the evaluation function of the alpha-beta search algorithm and can best be determined experimentally.

4.1.8 Avoiding self-destruction

In Khet it is possible for a player to destroy one of his own pieces. Usually, these are very bad moves that a player will never do. There might be some hypothetical situations where a player might want to hit one of his own pieces, but if they exist they are extremely rare.

During the alpha-beta search, self-destruction moves occur regularly. Even though in practice such moves will never be performed, they are still fully investigated. However, by avoiding self-destruction, all moves where a player captures one of his own pieces are ignored. They are not investigated, because it can be assumed that such move is never part of the perfect-play strategy of the player. As a result, large subtrees are pruned, so the number of nodes that have to be investigated is decreased.

4.1.9 Endgame databases

In order to speed up the search at the end of the game, an endgame database can be used. An endgame database contains the game-theoretical values (win, loss or draw) for a large number of board positions in the late game. Whenever in the search tree a board position is reached which is stored in the database, the stored value can be used and no further search is needed. Endgame databases work only well with convergent games, since these have a limited number of endgame positions. Since Khet is a convergent game, an endgame database can in principle be used.

There can be some trouble, however, if the number of endgame positions is too large. The absolute minimum number of pieces on the board is 6: two silver djeds, two red djeds and the two pharaohs. These pieces can be placed roughly on the following number of ways:

$$n_{end} = \binom{70}{1} \cdot 2^2 \binom{69}{2} \times \binom{67}{1} \cdot 2^2 \binom{66}{2} = 3.7761 \cdot 10^{11}$$

If we would use one bit per position, the size of the endgame database for these six pieces would be 43.96 GB. On modern computers, this is feasible. The problem, however, is that board positions with the absolute minimum number of pieces are very rare. If we would only store these positions, the endgame database will rarely be used. If we would extend the database with all positions where both players also have one pyramid, the number of possible board positions rises to more than $2.4 \cdot 10^{16}$. Such a database would require more than 22 million GB harddisk space while using one bit per position. It is obvious that this is infeasible. As a conclusion, endgame databases will not be used in Khet.

4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [6] is a technique that can be used to play a large variety of games [5]. 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*.

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 are carried out.

After the simulations are done, the algorithm chooses the move which has the highest win rate. The win rate is the number of times the move eventually resulted in a win for the player, divided by the number of times the move has been played. The move with the highest win rate is considered to be the best move to play and is returned by the algorithm.

This bare Monte Carlo algorithm can be extended in multiple ways. Often, a game 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 will investigate is called Upper Confidence bounds applied to Trees. This technique was succesfully used in the Go-playing-program MoGo, which has won multiple large Computer Go tournaments [26].

4.2.1 Upper Confidence bounds applied to Trees

The basic Monte Carlo algorithm selects a move completely randomly while simulating a game. The disadvantage of this is that bad, unlikely moves are played about as often as good, likely ones. By implementing Upper Confidence bounds applied to Trees (UCT) [16], a best-first search algorithm, it is possible to let the algorithm play better moves more often than bad ones.

At each node of the tree, two values are stored. The first is the number of times the node has been visited. The second is the number of times a game passing this node resulted in a win.

While simulating a game, after each move the algorithm checks whether there already exists a node in the tree that represents the new board position. If this is the case, then the UCT value for all children is calculated. The UCT value is calculated using the following formula:

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

This formula is described in [23]. It is a slightly modified version of the UCT formula described in [8].

Here, n is the current node and c is the child. w_x stands for the number of wins for node x and p_x for the number of times node x has been visited. C is a constant value, which determines the exploration/exploitation trade-off. In other words, it determines to which degree the algorithm uses past results to determine which move to choose and how often the algorithm should explore alternative moves. A lower value of C causes more exploration, while a higher value of C causes the algorithm to exploit the results earlier. The best value of C should be determined experimentally.

After calculating the UCT value for 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. This is because most of these nodes will never be visited again.

If enough samples are played, the algorithm will eventually converge to the best move.

4.2.2 Transposition tables

Similar to an alpha-beta 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 and the number of eventual wins is stored. These are the data that were stored in the search tree. Also the hash key is stored, which is used for type-2 error detection. This is similar to the transposition table used for alpha-beta search. Finally, the move number is stored. This number is used for the replacement scheme. This is slightly different from the transposition table for alpha-beta search, where the depth is stored. Whenever a collision occurs, the node with the lowest move number is stored (or kept) in the table, because this node has the highest chance to be visited again.

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.

Because in a transposition table nodes can have multiple parents, it is possible that the sum of the number of visits of the child nodes exceeds the number of visits of the current node. Some nodes might have been visited via other parents. The result of this is that these nodes will have a considerably smaller chance to be chosen, but since they can be reached by multiple parents, they will still be played often enough.

4.2.3 Maximum game length

Games of Khet can take a long time to finish. Random games can take more than 1000 moves before either player wins or the game ends in a draw. 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 needed.

Another way is to do a quick heuristic evaluation of the board. If one of the players has a clear advantage over the other, then this player can be declared the winner. This can make the results more useful, but such a heuristic evaluation takes some time to be performed. Also, some extra domain knowledge is needed to make a reliable estimation.

In our program we will only use the 'draw heuristic'. Future research will be necessary in order to determine how domain knowledge can best be used to improve the MCTS algorithm.

4.3 Move Generation

One of the most important functions of the program is the function that generates a list of all valid moves for a player in a given board position. Generating a list of all valid moves for a player is not a difficult task in Khet. The program keeps track of a list of the locations of the pieces for both players. For each piece, a list of possible moves is generated. For each of the eight possible target locations of the piece, a check is performed whether it is a legal move. Also, the rotation moves for the pyramids and the djeds are added. Then, the lists of possible moves for all pieces are combined and returned as the list of all possible moves for the player.

There are some variations on this move generation algorithm. One of them is Incremental Move Generation, which has been explained in Section 4.1.3. Another variation is the move generation algorithm used by quiescence search.

As explained in Section 4.1.4, quiescence search only needs capture moves. Even though it is impossible in Khet to determine beforehand which moves will be capture moves and which will not, it is often possible to filter out a fair amount of moves that can never be capture moves.

If we have a situation where, if the player would not perform a move, but just fire his laser, no piece is captured, then we know that all moves that do not change the course of the laser are not capture moves. This means that there are only two types of moves that can be capture moves:

1. All moves of the pieces that are within the course of the laser.
2. The moves of the pieces that are outside the course of the laser and move inside.

All other moves can never be capture moves, and thus they can be filtered out.

If we have a situation where the player would capture a piece without moving, then all moves can be capture moves. The moves that do not change the course of the laser are capture moves for sure, but also the moves that do change the course of the laser can be capture moves, although we cannot be sure.

4.4 Draw Detection

In Khet it is a rule that whenever a position occurs for the third time, the game can be declared a draw. In order to detect a draw, the program needs to keep track of the history of the game. This can be done in several ways. The first one is saving all board positions that occurred in the past. By comparing the current position to the past ones, the program can check for a draw.

Another way is by saving Zobrist keys instead of complete board positions. This introduces the chance of a type-1 error. It is possible that a new board position is detected to be a draw because in the past different board positions occurred with the same Zobrist key. However, the chance that this happens is negligibly small when using a 64-bit key. This method can be improved by emptying the history whenever a piece is removed from the board. This may be done, because whenever a piece is removed from the board, a past position cannot occur anymore in the future.

This technique is much more efficient than the first one. The program does not need to store the board positions, but only the 64-bit values, which makes it more space-efficient. Checking for a draw is also more effecient, since the program only needs to compare the Zobrist values, instead of the boards square-by-square. This means that this method is also more time-efficient.

The third method is the use of a hash table. It looks like a transposition table, but works differently. Just like with a transposition table, the table entry of a board position is determined by its Zobrist value. In each entry, the hash key is stored, along with the number of occurrences of the current position in the game. Again, a type-1 error can occur, but the chance of this happening is extremely small. It is even possible to throw away the hash key, because even a type-2 error has a very small chance of occurring.

This technique is less space-efficient, but it is much more time-efficient. The program does not need to count how many times the current board position has occured in the past, because this number is stored in the table.

Since we want this function to be as fast as possible, since it will be called at each node in the search tree, we will use the hash table technique in our program. In order to save memory, the hash key will not be stored.

Chapter 5

Experiments and Results

In this chapter a description of all experiments that have been performed is given. Also, the results of these experiments are presented. This chapter also provides some conclusions that can be drawn from these results. The general conclusions and the answers to the research questions are given in Chapter 6.

5.1 Alpha-Beta Search

In this section, we will give a description of the experiments concerning the alpha-beta search technique that were performed. Some experiments are performed to determine the best parameter settings for the aspiration search and quiescence search improvements. Also, the quality of all improvements is determined. This is done for each improvement by letting an alpha-beta player with the improvement play against an alpha-beta player without improvements, in order to check whether the improvement does make the player play stronger. For all these experiments where two players play against each other, both players receive 300 seconds to play the whole game. There is no maximum time per move. All games start with the Original set-up.

Another type of experiment is checking whether an improvement reduces the number of nodes that are investigated. By comparing the number of investigated nodes with and without improvements, we can conclude whether an improvement works the way it should and to what extent the improvement reduces the size of the search tree.

5.1.1 Quiescence search

For quiescence search, there is no use in finding out how many nodes are investigated at each ply. This is because, contrary to for example transposition tables or killer moves, more nodes are investigated instead of less. Therefore, we will only let an alpha-beta player with quiescence search play against an alpha-beta player without improvements. The results are displayed in Table 5.1.

As Silver			As Red			Total score
wins	draws	losses	wins	draws	losses	(out of 50)
4	0	21	5	0	20	9

Table 5.1: Experimental results for alpha-beta search with quiescence search versus bare alpha-beta search.

It is obvious from these results that quiescence search causes the player to play much worse. There are several reasons for why the basic quiescence-search algorithm does not work with Khet.

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. If this is not the case, then we have found a quiet move and the move has to be undone, since it will not be investigated. Even when filtering out all moves that can never be a capture move, this causes a large amount of overhead, since all the time spent on moving the piece and determining the path of the laser is wasted.

Another problem is that the branching factor can be very high. In some occasions, almost all of a player's moves are capture moves. Consider the situation depicted in Figure 5.1. This position can be reached from the Original set-up, after the moves *1. d6c6 h4g4*. In this position, Silver has 80 possible moves, 78 of which are capture moves (only *2. j4j3* and *2. j4r-* are not). Even when not counting the moves where Silver hits one of his own pieces, there are still 72 capture moves left. Since Silver will hit Red's obelisk at *g8* even without moving, all moves that do not change the course of the laser are, by default, capture moves. Situations like these occur regularly and cause the average branching factor during quiescence search to be relatively high.

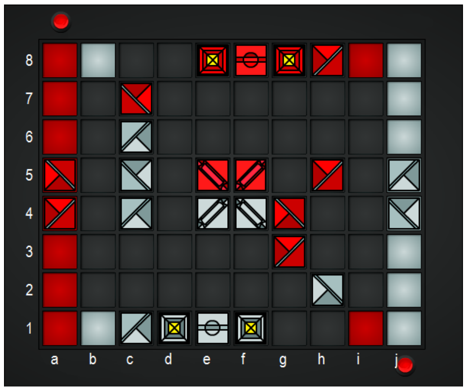


Figure 5.1: Position after *1. d6c6 h4g4*.

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 setup, subtrees are built that reach depths of 16 ply or more. For example, one of the paths in the tree reads: *1. c5c6 h4g5xc4 2. e4e3xg5 e5f6xd6 3. c1b1xh5 f8f7xc6 4. f4e4xe8 f5rxf1 5. j5r+xe8 f6e7xf1 6. h2h3xc7 f5g4xj5 7. e4d5xg8 e7d8xd1 8. e3f4xg8 g4g5xd1 9. f4rxf7, 1-0.*

Here, Silver wins after a sequence of 16 consecutive capture moves. This is only one of many complete games that are built when applying quiescence search to a 2-ply alpha-beta search.

Q_n-limited search

The previous results show that basic quiescence search does not work well in Khet. Therefore, we used Q_n-limited search. We performed experiments in order to determine the best value for n. We let an alpha-beta player with Q_n-limited search play against an alpha-beta player without improvements, with various values of n. The results are displayed in Table 5.2.

It is clear that Q_n-limited search can be a considerable improvement to the alpha-beta search algorithm. If the value of n is set to 2 or 3, the algorithm works much better than bare alpha-beta search.

	As Silver			As Red			Total score
n	wins	draws	losses	wins	draws	losses	(out of 50)
1	12	0	13	12	1	12	$24\frac{1}{2}$
2	20	2	3	14	0	11	35
3	19	0	6	15	0	10	34
4	12	1	12	9	0	16	$21\frac{1}{2}$
∞	4	0	21	5	0	20	9

Table 5.2: Experimental results for alpha-beta search with Q_n search for various values of n versus bare alpha-beta search.

5.1.2 Transposition tables

The main goal of transposition tables is to reduce the number of nodes that are investigated. This goal is achieved by detecting transpositions and by using move ordering. We can determine the usefulness of a transposition table by determining the number of nodes that are investigated while using a transposition table compared to the number of nodes that are investigated without a transposition table. Table 5.3 shows how many nodes are investigated with several search depths by the Silver player on the Original starting position.

These results show that for a search depth of 4 or more, the transposition table causes a considerable reduction of the number of investigated nodes.

A similar experiment can be done with an endgame position. Figure 5.2 shows an endgame position with the minimum number of pieces possible. Table

Ply	Without TT	With TT	% gain
1	80	80	0.00
2	1,388	1,349	2.77
3	39,451	37,099	5.96
4	803,056	193,160	75.95
5	22,431,125	3,949,921	82.39

Table 5.3: Number of nodes investigated at different search depths on the Original starting position with and without transposition table.

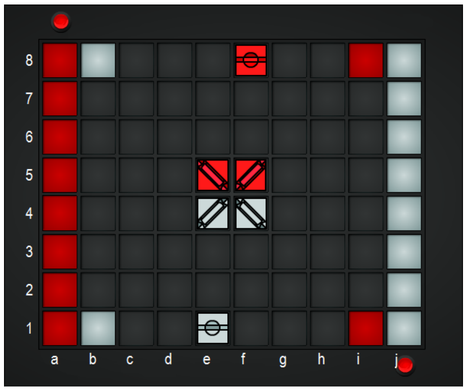


Figure 5.2: Example endgame position.

5.4 shows how many nodes are investigated by Silver on this endgame position with several search depths.

Again, the transposition table causes a reduction of the number of investigated nodes with a search depth of 4 or more, though the gain is not as large as on the opening position. The reason for this is obvious. In this situation, there is no clear best move. Because of the random factor, in each iteration another move turns out to be the best one. This makes the best move that is stored in the transposition table useless, so there is no gain here. The only gain comes from the transpositions in the search tree.

We can repeat these experiments with the random factor in the evaluation function turned off. This will cause the first move always to be the 'best' one. The results are displayed in Table 5.5 (opening position) and Table 5.6 (endgame position).

We can see that the number of nodes is drastically reduced now, especially for the endgame position.

Now that we have seen how transposition tables can reduce the number of nodes, we will show that a player with a transposition table plays stronger than a player without. Table 5.7 shows that, surprisingly, a player with only a transposition table does not play stronger than a bare alpha-beta player. The reason for this is the horizon effect. If a player can search deeper, it does

Ply	Without TT	With TT	% gain
1	18	18	0.00
2	183	182	0.76
3	2,065	1,928	6.63
4	21,302	15,696	26.31
5	197,052	116,829	40.71
6	1,897,196	981,053	48.29
7	10,249,628	5,222,414	49.05

Table 5.4: Number of nodes investigated at different search depths on the example endgame position of Figure 5.2 with and without transposition table.

Ply	Without TT	With TT	% gain
1	80	80	0.00
2	278	286	-2.88
3	18,102	16,363	9.61
4	50,246	31,901	36.51
5	10,573,712	2,213,587	79.07
6	65,615,083	14,473,885	77.94

Table 5.5: Number of nodes investigated at different search depths on the Original starting position with and without transposition table with the random factor turned off.

Ply	Without TT	With TT	% gain
1	18	18	0.00
2	52	52	0.00
3	413	411	0.48
4	1,100	1,040	5.45
5	8,648	6,973	19.37
6	23,401	16,517	29.42
7	238,076	91,660	61.50
8	635,364	210,622	66.85
9	6,277,169	1,026,730	83.64
10	31,886,010	2,288,274	92.82
11	451,894,920	11,733,500	97.40
12		42,928,527	
13		292,704,813	

Table 5.6: Number of nodes investigated at different search depths on the example endgame position of Figure 5.2 with and without transposition table with the random factor turned off.

	As Silver			As Red			Total score
	wins	draws	losses	wins	draws	losses	(out of 50)
Without Q_2-limited	14	0	11	10	1	14	$24\frac{1}{2}$
With Q_2-limited	17	1	7	14	0	11	$31\frac{1}{2}$

Table 5.7: Experimental results for alpha-beta search with a transposition table versus bare alpha-beta search, with and without Q_2-limited search enabled for both players.

not necessarily mean that the results are better, due to the horizon effect. If we repeat this experiment, but now with Q_2-limited search enabled for both players, we can see that the transposition table now turns out to be a considerable improvement.

5.1.3 Killer moves

The strength of the killer moves can, just as with transposition tables, be determined by comparing the number of nodes investigated when using killer moves to the number of nodes that are investigated without killer moves. The results are given in Table 5.8.

Ply	Without KM	With KM	% gain
1	80	80	0.00
2	1,388	1,394	-0.00
3	39,451	19,152	51.45
4	803,056	78,035	90.28
5	22,431,125	887,925	96.04
6		4,317,365	
7		65,626,000	

Table 5.8: Number of nodes investigated at different search depths on the Original starting position with and without killer moves.

From these results it is obvious that killer moves cause a huge decrease of the number of investigated nodes.

The results of the matches of alpha-beta search with killer moves versus an alpha-beta player without killer moves are displayed in Table 5.9. It is obvious that killer moves cause a very large improvement of the strength of the alpha-beta player. It turns out that, contrary to transposition tables, it does not even need the presence of Q_2-limited search in order to cause a great improvement of the performance of the alpha-beta player.

5.1.4 Aspiration search

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.

	As Silver			As Red			Total score
	wins	draws	losses	wins	draws	losses	(out of 50)
Without Q_2-limited	21	0	4	20	1	4	$41\frac{1}{2}$
With Q_2-limited	20	2	3	21	0	4	42

Table 5.9: Experimental results for alpha-beta search with killer moves versus bare alpha-beta search, with and without Q_2-limited search enabled for both players.

For these experiments, we enable Q_2-limited search 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_2-limited search, these fluctuations are decreased.

From the results in Table 5.10, we can conclude that aspiration search works best with $\Delta = 500$. However, aspiration search only makes the alpha-beta player play modestly better.

	As Silver			As Red			Total score
Δ	wins	draws	losses	wins	draws	losses	(out of 50)
1	10	4	11	11	1	13	$23\frac{1}{2}$
100	11	2	12	11	1	13	$23\frac{1}{2}$
500	14	2	9	13	1	11	$28\frac{1}{2}$
1000	12	1	12	12	1	12	25

Table 5.10: Experimental results for alpha-beta search with aspiration search with various values of Δ, and Q_2-limited search versus alpha-beta search with only Q_2-limited search.

5.1.5 Summary

In the previous sections, we have given the experimental results for the transposition table, killer moves, quiescence search and aspiration search. In Table 5.11, all these results, along with the results for incremental move generation and avoiding self-destruction, are summarised.

	As Silver			As Red			Total score
Improvement	wins	draws	losses	wins	draws	losses	(out of 50)
Incr. move generation	13	0	12	14	0	11	27
Q_2-limited search	20	2	3	14	0	11	35
Transposition table ($+Q_2$)	17	1	7	14	0	11	$31\frac{1}{2}$
Killer moves ($+Q_2$)	20	2	3	21	0	4	42
Aspiration search ($+Q_2$)	14	2	9	13	1	11	$28\frac{1}{2}$
Avoiding self-destruction	10	2	13	10	1	14	$21\frac{1}{2}$

Table 5.11: Experimental results for the alpha-beta algorithm with single improvements versus bare alpha-beta search.

These results show that the win rates of incremental move generation, aspiration search and avoiding self-destruction are less than 60%, for the last one even less than 50%. From these results we can conclude that these improvements do not cause the alpha-beta player to play much stronger. They can even cause the player to play worse. 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.

Quiescence search (in the form of Q_n-limited search), transposition tables and killer moves are good improvements. These three improvements cause a considerable increase of the strength of the alpha-beta player.

5.2 Monte Carlo Tree Search

This section describes the experiments and results for the Monte Carlo Tree Search algorithm. We will investigate the strength of UCT and whether or not setting a maximum game length causes the MCTS player to play stronger. The experiments are similar to those for the alpha-beta player. We will let an MCTS player with one improvement play against an MCTS player without improvements. The settings are also the same: again both players receive 300 seconds to play one game and all games start with the Original setup.

5.2.1 Upper Confidence bounds applied to Trees

As explained in Section 4.2.1, the formula for calculating the UCT value of the children of a node contains a constant factor C. In order to determine the best value of C and to test the performance of UCT, we let an MCTS player with UCT play against a bare Monte Carlo player without UCT, with different values for C. The results of these experiments are summarised in Table 5.12.

From these results, we can conclude that UCT causes a great performance improvement of the MCTS player. UCT works best with $C = 7$.

	As Silver			As Red			Total score
C	wins	draws	losses	wins	draws	losses	(out of 50)
1	17	0	8	16	0	9	33
3	24	0	1	20	0	5	44
5	23	0	2	21	0	4	44
7	24	0	1	25	0	0	49
9	23	0	2	20	0	5	43
11	24	0	1	21	0	4	45
13	21	0	4	24	0	1	45
15	21	0	4	18	0	7	39

Table 5.12: Experimental results for MCTS with UCT with various values for C versus bare Monte Carlo.

5.2.2 Maximum game length

In order to prevent the Monte Carlo algorithm from playing endlessly long games, we can define a maximum game length. After a certain number of moves, d_{max}, an evaluation function is called to determine a score for the game. In our program, the evaluation function returns a score of 0.5, denoting a draw.

In order for this improvement to work as good as possible, we need to find the best value of 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. With $d_{max} = \infty$, a game is never cut off and the improvement has no effect.

The results of the MCTS players with different values of d_{max} are displayed in Table 5.13. These results show that the MCTS player plays best with $d_{max} =$ 100.

	As Silver			As Red			Total score
d_{max}	wins	draws	losses	wins	draws	losses	(out of 50)
25	14	0	11	11	0	14	25
50	14	1	10	17	0	8	$31\frac{1}{2}$
100	17	0	8	19	0	6	36
200	15	0	10	17	0	8	32

Table 5.13: Experimental results for MCTS with various values for d_{max} versus MCTS with $d_{max} = \infty$.

5.3 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. For the alpha-beta player, we use a combination of a transposition table, killer moves, Q_2-limited search, incremental move generation, and aspiration search (with $\Delta = 500$). The MCTS player uses UCT with $C = 7$ and a maximum game length $d_{max} = 100$. All games are played starting with the Original set-up. The results are displayed in Table 5.14.

	As Silver			As Red			Total score	
Time	wins	draws	losses	wins	draws	losses	Alpha-beta	MCTS
60	50	0	0	50	0	0	100	0
300	25	0	0	25	0	0	50	0
1800	5	0	0	5	0	0	10	0

Table 5.14: Experimental results for alpha-beta search versus MCTS

From these results, it becomes immediately clear that alpha-beta search works much better than MCTS. The alpha-beta player wins all games. It turns out that because of the lack of strategic domain knowledge, MCTS does not stand a chance. Even the small amount of extra domain knowledge makes the alpha-beta player play on a much higher level than the MCTS player.

One example game between alpha-beta search and MCTS, along with a detailed explanation, can be found in Appendix A. Using this example game, we will show some of the strengths and weaknesses of both players.

Chapter 6

Conclusions

In this chapter, we will give answers to the research questions and the problem statement given in Section 1.3. Moreover, we will give an overview of possible subjects for future research.

6.1 Answering the Research Questions

In Section 1.3, we have defined four research questions stemming from the problem statement. The answers to these research questions can be found in the previous chapters. In this section, we will revisit the research questions and summarise the answers.

What is the game complexity of Khet?

This research question has been answered in Chapter 3. The state-space complexity is, according to our calculations, 10^{48}. The game-tree complexity of Khet is 10^{124}. These numbers are only rough approximations. Especially the value of the game-tree complexity is subject to change as more information about Khet becomes available. This value is only based on a relatively small number of games between two basic alpha-beta players. If more research is done, this value may be revised.

With these numbers we can conclude that the complexity of Khet is very close to the complexity of chess. This means that the game is, in the near future at least, unsolvable, but it should be possible to create a computer player that can challenge the world's best players.

Which search techniques can be used to play Khet?

In general, there are two different search techniques that have been investigated during this research: alpha-beta search and MCTS. Alpha-beta search is a strong and highly expandable depth-first search algorithm which works for a large variety of games. MCTS is a Monte-Carlo-based search algorithm which also works well for a large variety of games, and which does not need strategic knowledge.

How can these search techniques be improved to increase the strength?

In Chapter 4, we have defined a number of possible improvements for both alpha-beta search and MCTS.

For alpha-beta search, most improvements are concerned with reducing the number of nodes that have to be investigated during the search. Killer moves, transposition tables, avoiding self-destruction and aspiration search are examples of improvements that try to achieve this. An endgame database can be used to store the values of endgame positions, so the search can be terminated whenever a position is reached in the search tree that is stored in the endgame database, thus reducing the number of investigated nodes. Another way of improving the search is by increasing the number of nodes that can be investigated per second. Incremental move generation is a technique that can do this. Quiescence search, including a variation called Q_n-limited search, does not try to increase search speed or reduce the number of nodes investigated, but it extends the search tree in a smart way to improve the quality of the results.

Which combination of techniques has the best balance between speed and strength?

In order to determine how well the techniques and improvements work, we have performed a number of experiments that are described in Chapter 5. Among the tested improvements for alpha-beta search, the combination of a transposition table, killer moves and Q_2-limited search gives the best results. MCTS works best with UCT enabled and with $C = 7$ and $d_{max} = 100$.

According to the results of the final experiments, the alpha-beta search algorithm works far better than MCTS.

6.2 Answering the Problem Statement

Now that we have answerd the research question, we can formulate an answer to the problem statement that we have given in Section 1.3.

Is it possible to develop a computer program to play the game Khet effectively and efficiently?

Yes, it is possible to create a computer program that can play Khet efficiently and effectively. The complexity of Khet lies very close to the complexity of chess, which means that it will not be possible to solve the game now, or in the near future, but it should be possible to play the game at the level of a world champion.

Alpha-beta search turns out to be the best algorithm to play the game. A transposition table, killer moves and quiescence search (in the form of Q_2-limited search) are significant improvements to the alpha-beta algorithm that let the player play stronger.

Although it is difficult to judge the overall strength of our program due to the lack of artificial and human opponents, based on the experiments our educated guess is that the program plays Khet at a reasonable level, though there is still a lot of room for improvement.

6.3 Future Research

This research is the first one for the game Khet. This means that there is still a lot of possible future research.

One thing that really needs improvement is the evaluation function of the alpha-beta algorithm. The strength of the alpha-beta player relies heavily on the quality of the evaluation function and currently the evaluation function is pretty basic. This causes the alpha-beta player to follow the basic Khet strategies, but that will not be enough to challenge the world's best players. More strategic knowledge will be necessary to increase the strength of the alpha-beta player.

Strategic knowledge can also improve the performace of the MCTS player. It is, however, questionable whether MCTS will ever match the performance of alpha-beta search. Currently, the difference is very large, and, considering the complexity analysis, alpha-beta search is likely to remain the strongest search algorithm.

An addition that is used in many games, but is still missing in our program, is an opening book. In Khet this is tricky, since there is no official starting position. This means that for each set-up a different opening book needs to be created. Games with the same set-up often start in a similar way and the implementation of an opening book 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 all-important endgame. Unfortunately, such opening books are not yet available.

Appendix A

An Example Game

Here, we will show one of the games between alpha-beta and MCTS. This game comes from the last set of experiments, where both players receive 1800 seconds to play a game. In this example game, the alpha-beta player plays as Silver and the MCTS player plays as Red.

Silver starts with *1. j4j3* (see Figure A.1). This is generally a good move to start with, as he puts immediately pressure on the red pyramid at *h4*. This piece can be captured by Silver by either *2. j3j2* or *2. h2h3*. Possible ways for Red to defend this pyramid are *1. ... g3h3* or *h4r+*. Unfortunately, Red answers with *1. ... g3g4* (see Figure A.2). The MCTS player does not care about losing a piece, as this strategic domain knowledge is not implemented for this player.

Red can capture the pyramid on *h4* now, but instead chooses *2. f4e3xg4* (see Figure A.3). This is an even better choice, since there is no way Red can defend his pyramid at *h4* anymore. Instead of trying to limit the damage, Red responds by capturing one of his own pyramids by the move *2. ... a5b4xa4* (see Figure A.4). The reason for this becomes clear after the next two turns: *3. e3f3xh5 b4a3 4. h2h3xh4 a3a2* (see Figure A.5).

While Silver captures two more of Red's pyramids, Red moves his pyramid down trying to put pressure on Silver's pharaoh. At least, Red has now put pressure on the silver pyramid at *c4*, but Silver responds with *5. c4r-*, preventing Red from capturing this pyramid. Red is not able to capture one of Silver's pieces and performs the move *5. ... g8uf7* (see Figure A.6). Apparently, this looks a little bit safer, according to Red.

The game continues as follows: *6. c4b3 h8g7 7. e4d5 e8e7 8. d5d6 e5f6 9. d6e7 f5e6* (see Figure A.7). Silver has moved one of his djeds from *e4* to *e7*, putting it very close to Red's pharaoh. Also, he moved the red stacked obelisk from *e7* to *d6*, reducing its value. Meanwhile, Red has improved his defenses. He even set up a booby-trap with his djeds. If Silver is not careful, he might hit his own pharaoh. This opportunism is very characteristic for the MCTS player. However, an alpha-beta player will never make this mistake.

Next, Silver turns up the pressure with the moves *10. d5r+ c7d7 11. j5r-*

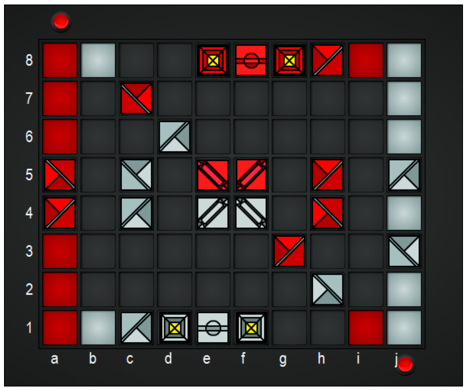


Figure A.1: Position after *1. j4j3.*

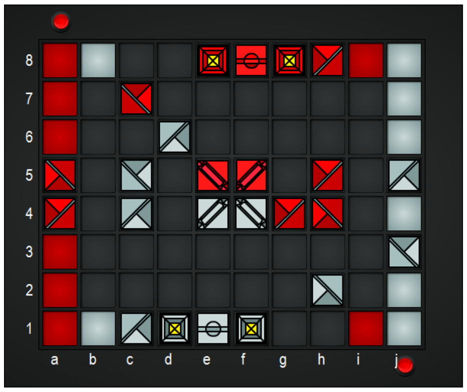


Figure A.2: Position after *1. ... g3g4.*

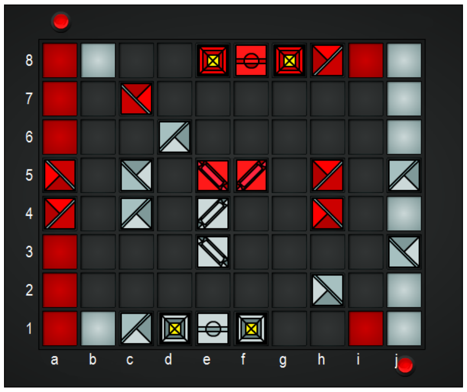


Figure A.3: Position after *2. f4e3xg4.*

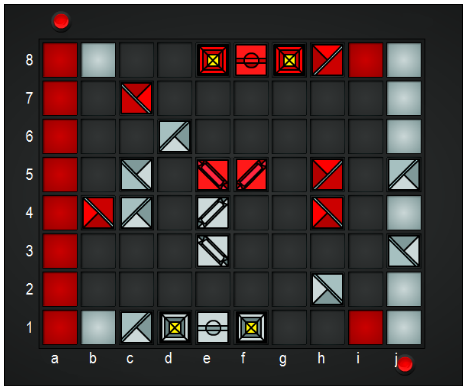


Figure A.4: Position after *2. ... a5b4xa4*.

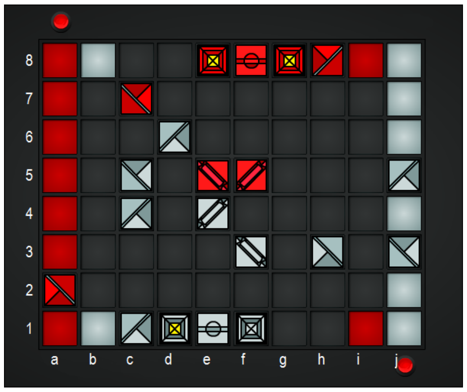


Figure A.5: Position after *3. e3f3xh5 b4a3 4. h2h3xh4 a3a2*.

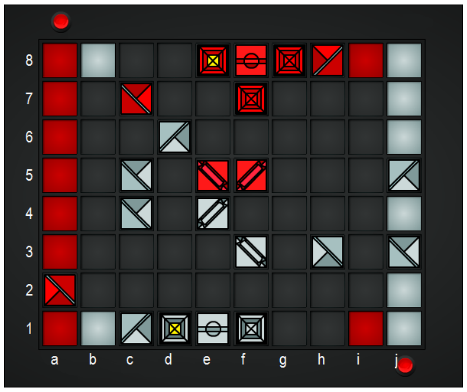


Figure A.6: Position after *5. c4r- g8uf7*.

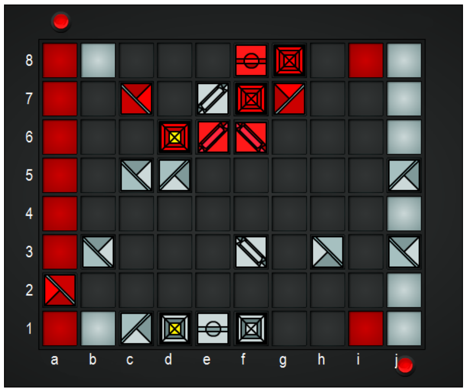


Figure A.7: Position after *6. c4b3 h8g7 7. e4d5 e8e7 8. d5d6 e5f6 9. d6e7 f5e6.*

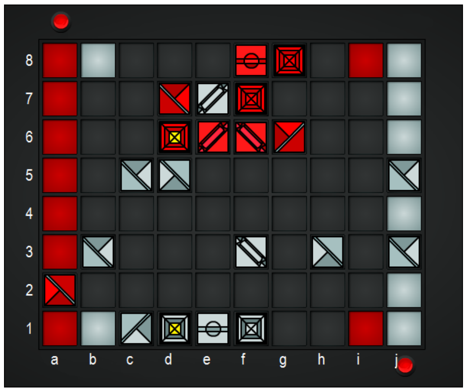


Figure A.8: Position after *10. d5r+ c7d7 11. j5r- g7g6.*

g7g6 (see Figure A.8). Now, Silver can move the pyramid at *j3* out of the route of the laser beam, in order to capture one of the red obelisks at *d6*. This might have been Silver's plan before Red moved his pyramid from *g7* to *g6*. But now, Silver chooses another strategy and plays *12. j5i6* (see Figure A.9). Apparently, Silver now wants to use this pyramid and the pyramids at *h3* and *j3* to try to attack Red's pharaoh. The djed at *e7* now also might come in handy. Red should react by returning the pharaoh from *g6* back to *g7*, but plays *12. ... d6c7* (see Figure A.10). Now Silver can easily finish the game.

Silver keeps moving his pharaoh forward and Red does not even try to defend his pharaoh anymore. He could try to use his djeds to delay his defeat, but he does not. The game continues with the moves *13. i6h7xf7 g8h8 14. h7g8xh8* (see Figure A.11).

Now, there is no way for Red to defend his pharaoh anymore. He plays the completely random move *14. ... c7ub7* and Silver finishes with *15. g8h8xf8 1-0.* It is a rather short game, but it shows some interesting characteristics of both

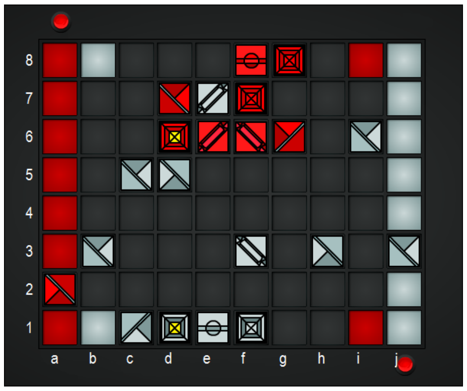


Figure A.9: Position after *12. j5i6.*

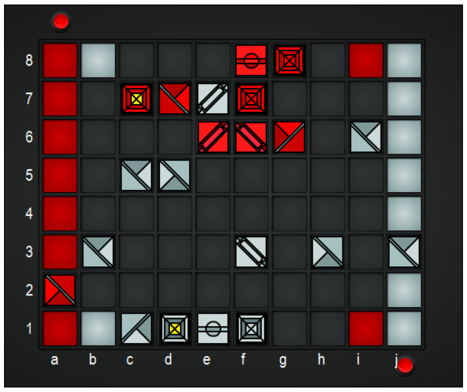


Figure A.10: Position after *12. ... d6c7.*

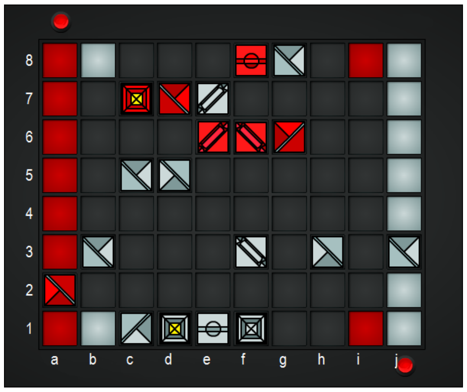


Figure A.11: Position after *13. i6h7xf7 g8h8 14. h7g8xh8.*

the alpha-beta player and the MCTS player.

Bibliography

[1] V. Allis. *A Knowledge-based Approach of Connect-Four.* Master's Thesis, Vrije Universiteit, Amsterdam, The Netherlands, 1988.

[2] D. F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43(1):85–98, 1990.

[3] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *ICCA Journal*, 19:175–180, 1996.

[4] A.L. Brudno. Bounds and valuations for abridging the search of estimates. *Problems of Cybernetics*, 10:225–241, 1963.

[5] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. Universiteit Maastricht / MICC, 2008.

[6] 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.

[7] R. Gasser. Solving Nine Men's Morris. *Computational Intelligence*, 12:24–41, 1996.

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

[9] R.D. Greenblatt, D.E. Eastlake III, and S.D. Crocker. The Greenblatt Chess Program. *Proc. AFIPS Fall Joint Computer Conference*, 31:801–810, 1969.

[10] A.D. de Groot. Het denken van den schaker, een experimenteel-psychologische studie. *Academic thesis, University of Amsterdam*, 1946. In Dutch.

[11] H.J. van den Herik. *Computerschaak, Schaakwereld en Kunstmatige Intelligentie.* Academic Service, Den Haag, The Netherlands, 1983. In Dutch.

[12] 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.

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

[14] Khet: The laser game, 2006. Online; `http://www.khet.com`.

[15] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[16] L. Kocsis and C. Szepesvri. Bandit based monte-carlo planning. *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.

[17] M. Minsky. *Semantic Information Processing.* M.I.T. Press, Cambridge, MA, USA, 1968.

[18] A. Newell and H.A. Simon. *Human Problem Solving.* Prentice-Hall Inc., Englewood Cliffs, NY, USA, 1972.

[19] N.J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* McGraw-Hill Book Company, New York, NY, USA, 1971.

[20] P.A. Piccione. In search of the meaning of Senet. *Archaeology*, July/August:55–58, 1980.

[21] J.W. Romein and H.E. Bal. Solving Awari with parallel retrograde analysis. *Computer*, 36:26–33, 2003.

[22] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, Second Edition.* Prentice-Hall Inc., Englewood Cliffs, NY, USA, 2003.

[23] Sensei's library: UCT. Online; `http://senseis.xmp.net/?UCT`.

[24] C.E. Shannon. Programming a computer for playing chess. *Philosophical Magazine*, 41:256–275, 1950.

[25] A.M. Turing. *Faster than Thought*, chapter Digital Computers Applied to Games. Sir Isaac Pitman, London, 1953.

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

[27] A.L. Zobrist. A hashing method with applications for game playing. *Technical Report 88*, 1970. Reprinted (1990) in *ICCA Journal*, 13(2):69-73.