Analysis and Implementation of the game Gipf

Diederik Wentink

October 3, 2001

Preface

Contents

List of Figures

List of Tables

Abstract

This is the abstract of my thesis

Samenvatting

Chapter 1

Introduction

1.1 Computer games

Board games have always fascinated people. Playing these games is a way to proof your intelligence being superior to your opponent's. For thousands of years mankind has spent a lot of time with this occupation.

In 1769 the first man came with the idea to have a machine play chess. Von Kempelen called his automaton "The Turk". This 'chess computer', which was actually a man playing chess in a machine, has been followed by a lot of ideas and implementations of automata playing board games.

Between 1833 and 1871 Babbage designed the Analytical Engines. Although he has never been able to actually create one of his Engines, he designed a calculator, a tic-tac-toe playing machine and he uttered the idea that any board game could be played by an analytical engine.

The first realized chess-playing machine was constructed by Torres y Quevedo around 1890. The KRK-automaton played an endgame with White having a king and a rook against Black having only a king. The algorithm Torres y Quevedo used was pure knowledge-based, opposed to the search algorithms that were used half a century later.

The first mathematical ideas for computers playing games originated in the beginning of the 20th century. Zermelo [38], König [20] and Euwe [9] developed mathematical propositions and rules for the recognition of a win, a loss or a draw in chess, before the actual final position was reached.

Von Neumann and Morgenstern developed the minimax algorithm (4.2) and the first formal game theory. They stated in [25] that it would in theory be possible to decide who would win the game of chess before a move was made, but realized that there was no practically usable method to determine it.

De Groot [12] developed a theory about 'the thinking of a chess player'. He took into account the psychological aspects of the way people play chess. This is contrary to the game research done earlier, which was more about the mathematical side of playing games.

In 1950 Shannon described three strategies a computer can follow when playing chess [33]. These three strategies were the foundations of chess research for following years. In the introduction of his article Shannon substantiates that chess is an ideal research domain. One of his arguments can be seen as the beginning of the artificial-intelligence research [16].

The Bernstein group implemented the first computer able to play a complete chess game in 1958. Some others followed, but with the invention of the α-β algorithm (see section 4.3) by Newell, Shaw and Simon [26] in 1958, the modern age of computer game playing started. The correctness of this algorithm was proved in 1975 by Knuth and Moore [19].

Samuel has been working on 8×8 checkers, instead of chess. He was the first to introduce machine learning in the game world in 1959 [29]. He developed methods to learn improvements of the evaluation function.

In the 1960s, developments followed each other quickly, which lead to the first computer-chess match in 1967 between the Stanford University and ITEP from Moscow. Although no new techniques were presented for this match, it accelerated the development of chess programs.

Since then a lot of fine tuning has been done to make computers play stronger. Next to better playing techniques, the hardware became millions of times faster. These two things led to the victory of the computer Deep Blue over the human world champion Kasparov in 1997, an indication that a computer can play chess at least as good as a human being.

This short history was mainly adapted from [16], pages 47 – 207, where a much more extensive description of the history of computers playing games can be found.

Chess has always been the most important game in AI research. However, recently research concerning other games has significantly increased. It appears that other games can be more interesting for experiments with certain techniques or approaches. Some games, e.g. Go-Moku, are cracked by a computer program [4, 17]. This means that the program is capable of achieving the best possible theoretical results whatever opposition it faces [3]. An even stronger result, a game being solved, holds for e.g. Connect-Four [36]. A game is solved when it is not only cracked, but also the moves it makes are explicable in human terms.

Another example of an interesting game is Go. Go is still a big challenge to AI researchers. This game requires a totally different approach from chess. The programs that are written are almost solely knowledge-based, opposed to the search-based chess programs. Nowadays, computer Go is only capable of playing at amateur level.

Some years ago a new game is invented by Kris Burm. It is called Gipf (to be pronounced as if in German); it is the first game of the Project Gipf. This game has some very interesting properties for having it played by a computer. The rest of this thesis will explain the game, the way it is implemented in a computer program and the results from the program.

1.2 Project Gipf

The game Gipf is the basis of a series of games. Gipf essentially is played with basic stones on a hexagonal board. This is the characteristic for all games in the Project Gipf. The purpose of the Project is to create a whole framework existing of different games, each game with some other rules and types of stones. The different stones in the games, called potentials, can be combined to create new games from the existing ones.

The potential of the first game, Gipf, is the Gipf stone. In section 2.1 the working of this potential is explained.

The games that have already been marketed are Gipf, Tamsk, Zèrtz and Dvonn. Each game has its own potentials. There are more games to follow. More information on these games can be found at [40]. In the magazine Abstract Games, some reviews of games from the Project [13, 14] and an interview with the inventer Kris Burm [15], have already been published .

1.3 Problem statement

The problem statement of this thesis is based on two research questions. The first question is:

> *What is the best approach to an implementation of the game of Gipf according to the complexity of the game?*

If the game-tree complexity and the state-space complexity are known, something can be said about the possibility to crack or solve the game [3, 17]. If it is not possible to crack the game, the complexity indicates whether a knowledge-based approach or a brute-force approach is recommended.

The second research question is stated as follows:

> *What knowledge do experienced players use in playing the game of Gipf and can this knowledge be used in an implementation of the game?*

The knowledge will be gathered by interviewing experienced players and have them explain their choices.

From these two research questions the following problem statement has been derived:

> *What techniques can be used to develop computer program that plays the game of Gipf in the tournament-version as strong as*

possible and how can the knowledge of experienced players be integrated in this implementation?

The program developed to test this problem statement has participated at the Computer Olympiad in August 2001 in Maastricht.

1.4 Outline of the thesis

After this introduction the rest of the thesis will start with an explanation of the game Gipf. In chapter 2 all the rules will be noted, the different versions of the game and some strategies will be pointed out.

In chapter 3 the complexity of the game is examined. This is split up into the state-space complexity and the game-tree complexity. These complexities will be compared to the complexities of other games and a conclusion will be derived about the best approach of programming Gipf, according to its difficulty.

In chapter 4, the different search techniques which are potentially good for the implementation of the game are discussed. The chapter will go into the basic search algorithms and the refinements to make the program play better.

A discussion of the implementation of the game is described in chapter 5. It will treat the internal representation of board and moves, the construction of the search tree, some words about the implementation of certain search techniques and an explanation of the use of strategies.

Further, in chapter 6 the results of the developed program are presented. The different techniques used have been tested and their results will be presented here.

This thesis will end in chapter 7 with some conclusions, derived from the research done to this interesting game. Furthermore, some suggestions for future research will be given.

Chapter 2

Gipf: The game

2.1 The rules

Gipf is a relatively new game which was invented by Kris Burm. The game is brought out by Rio Grande Games and Don & Co NV in 1996. Gipf is a two-player, zero-sum boardgame with perfect information. It is played at a hexagonal board with seven lines in each direction, surrounded by 24 *dots*, see figure 2.1. Within these dots lies the actual playing board; the cross points of the lines are the *spots* on the board. There are 37 spots. Each player (Black and White) starts with an equal number of stones between 15 and 18 in hand. The stones in hand are called the reserve.

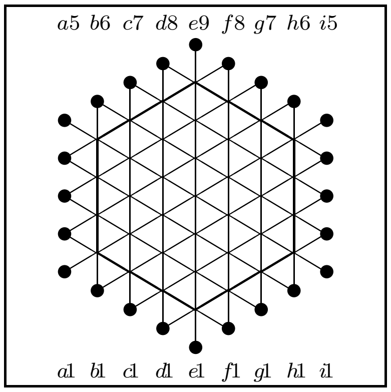


Figure 2.1: Empty Gipf board.

At every move a stone is put onto the board following some rules. Sometimes, stones come back from the board to the reserve. When a player has no stones in his reserve so he cannot move, he loses the game.

A move is made by shoving a stone from a dot (at the edge of the board) onto a spot adjacent to that dot. If there is a stone at that spot, it is pushed one spot further, in the direction of the move (along the line). If the spot where it is pushed to is occupied by another stone, then that stone is pushed further, etcetera, until an empty spot is reached.

If there is no unoccupied spot in the move direction, the move cannot be made because a stone would be pushed off the board, which is illegal.

At any time, when minimally four stones of one color lie next to each other at one line, without empty spaces between them, they have to be taken from the board and added to the reserve. When there are stones from the opposite color at the same line, next to these four stones, they are captured by the player with the four in a row. Always when a player has four stones in a row, all stones along that line are taken from the board, as long as there is no unoccupied spot between the stone and the four stones. The stones from the four-in-a-row color go back to the reserve, the others are captured.

It does not matter which one of the players causes the four-in-a-row. One player can force the other player to take his stones back.

It is possible that in one move, two or more four-in-a-rows are formed simultaneously. Moreover, four-in-a-rows may intersect. If there is an intersection between two four-in-a-rows of the same color, the player with that color decides which row to take. If they are from different colors, the stones are taken off in the order of playing. So, if White causes a white four-in-a-row intersecting a black four-in-a-row, then at first White has to take his stones from the board and after that, Black has to take them off in case there is still a black four-in-a-row.

Basic game

At the basic game, each player starts with 15 stones. Three stones from each player are put on the board at the starting position, see figure 2.2. So each player has 12 stones left in reserve. Both players move alternately until one of them has to move but has no stone left in reserve.

Standard game

At the standard game a new stone is introduced: the Gipf stone. This stone consists of two basic stones on top of each other. A Gipf stone takes two stones from the reserve of the player. If there is a Gipf stone in a four-in-a-row, the player who takes the stones from the board can decide whether to take the Gipf stone or leave it on the board. This applies to all Gipf stones in the row that is taken, irrespective their color.

The standard game starts with 18 stones for each player. At the starting position, three Gipf stones for each player are put at the board, at the same spots as the starting position of the basic game (see figure 2.3. So each player starts with 12 stones in reserve. From here the game is the same as the basic game. A player is not allowed to put new Gipf stones at the board.

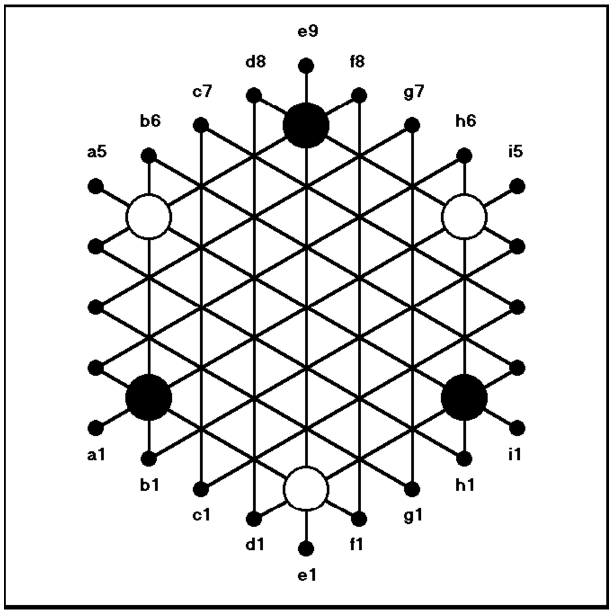


Figure 2.2: Starting position basic game

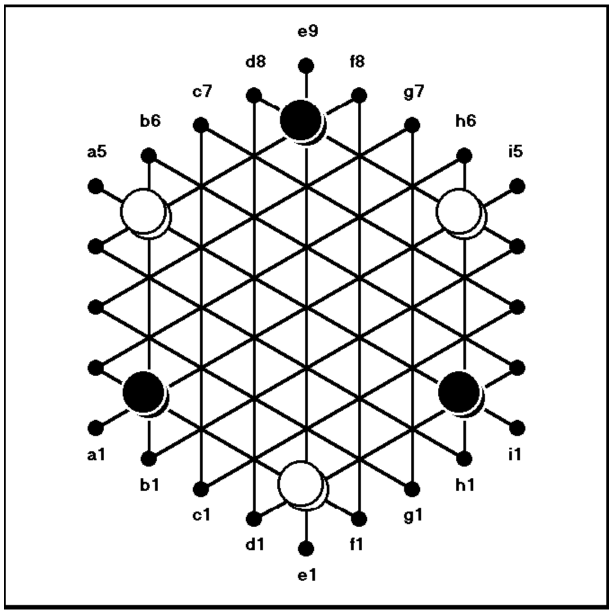


Figure 2.3: Starting position standard game

An important difference with the basic game is that there are two ways to win. The first one is the same as the basic game, when the opponent has run out of stones. The second one is when the opponent does not have any Gipf stone left at the board. So there are two strategies for playing: trying to let the opponent run out of stones and trying to capture all the opponents Gipf stones.

Tournament version

The tournament version is the ultimate Gipf game. It is played the same way as the standard game, with this important difference that the starting position is an empty board. Each player starts with playing a Gipf stone. As a next move he may choose whether he plays a Gipf stone or a basic stone. As long as a player has not played any basic stones, he is allowed to play Gipf stones. After he plays his first basic stone, he is not allowed to play any Gipf stone anymore. Whether one is allowed to play Gipf stones is independent from whether the opponent is allowed to play Gipf stones.

Like at the standard game, a player has won the game if either his opponent cannot move because he has run out of stones or the opponent does not have any Gipf stone left.

The clock

Usually in tournaments the game is played with a clock. Each player gets a fixed amount of time on the clock to finish the game (usually 20 minutes). If the flag falls for a player, he has lost the game.

Additional rules

Because Gipf is a very young game, the rules are not unambiguous. At tournaments there have been situations which were not properly covered by the rules, especially when a clock is used. In that case the referee decides what to do. Some situations below show their current solutions, but it is possible that solutions change in the future.

In principle a draw is not possible. However it has occurred that players keep repeating moves. At this situation a special draw rule has been applied. If both players have been repeating their move for three times, a player can ask for an interruption of the game. The referee stops the clock and the players start a new game, with the same colors, with the time left on the clock for both players.

When a player is making a move and during this move his flag falls, the player is allowed to finish his move within one minute. This rule is only of interest if a game is won in this last move. It does not matter whether the game is won by capturing the last Gipf stone of the opponent or it is won because the opponent has run out of stones.

2.2 Strategies and Knowledge

It is expected that the main strength of a computer program playing Gipf will come from search (see section 3.3). However, to play a good game, some knowledge has to be added to a program. Therefore knowledge has been acquired from the Eindhovense Gipf Organization (EGO). The members of this club are from the Technical University of Eindhoven and play a Gipf competition. They have explained some strategies, strong board positions and openings. Some of these strategies have been converted into algorithms and have been integrated into the computer program that has been developed (see section 5.3).

2.2.1 Centre stones

The first strategy concerns the centring of stones. This means that a player tries to get the stones in the more central positions of the board.

A stone always lies on three lines. At each of these lines, the stone can be part of a four-in-a-row. The longer a line is, the more ways there are to make a four-in-a-row at that line. If a stone is at the middle of a long line, there are many ways to use the stone in different four-in-a-rows. If a stone for example is lying at position b2, there are three ways to use this stone in a four-in-a-row, for each direction one. If a stone is lying at e5, exactly the centre of the board, there are 12 different ways to use this stone in a four-in-a-row. The more opportunities there are for a stone to be part of a four-in-a-row, the stronger the stone is.

This counts even more for Gipf stones. Once a Gipf stone lies in a good, central position, it is possible to keep it there for several moves.

A disadvantage of having Gipf stones more central is that there are also more ways to attack the Gipf stone. If a Gipf stone lies at the edge of the board it is very hard to attack it. There are only two ways to capture it (only through the two lines which are not edges). If the Gipf stone lies at the center spot, e5, there are six ways to push it away with a four-in-a-row and capture it. However, as long as the Gipf stone stays fixed at e5, it can not be captured.

The game can only be won if you play aggressive and try to capture stones from the opponent. A Gipf stone is very useful in an attack. If it is used for defense, it costs two stones to occupy a spot. In an attack, the threat from the Gipf stone remains after stones are taken of the board. From this point of view, the attacking power of a Gipf stone in the centre of the board is more important than the greater risk it runs to be captured.

2.2.2 Clustering stones

Stones of the same color are useful if they lie close enough to each other to be part of a four-in-a-row. If a player has several stones on a cluster, he often will be able to create a four-in-a-row in many directions, using only one stone. He only has to create a position where he can capture some stones from the opponent with the four-in-a-row.

A disadvantage of the clustering is that the opponent can easily create a situation where the player has to take back some of his stones, leaving a gap in the cluster which can be used by the opponent to capture stones.

Clustering Gipf stones is almost at any time an advantage. Gipf stones become stronger when they are lying next to each other at the same line, because one stone can never be a threat. When the Gipf stones are lying spread over the board, they can all be part of a threat with some basic stones. When stones are taken of the board, the threat is gone.

If the Gipf stones are lying in a cluster or in a line, the threat continues after stones are taken of the board, forcing the opponent to keep defending the threatened lines.

2.2.3 Dominating lines

A very important strategy the players from Eindhoven use, is the strategy of dominating lines. The main object of the players is to block the three longest lines in the line direction (from e2 to e8, from b2 to h5 and from b5 to h2) by occupying all spots at these lines. They try to have more stones in the lines than the opponent. When for example White has four stones in the same blocked line and Black has three stones, White is said to be dominating that line.

After a player dominates a line, he tries to move a stone onto the line from the side, causing a four-in-a-row for him and loss for the opponent. Because the player has to take stones from the board, he is likely to lose domination in other lines, or even give the opponent the opportunity to win stones in a line he dominates. The player who foresees best what is going to happen after he takes a four-in-a-row is likely to be in the strongest position some moves further on.

An example is given in figure 2.4. Here White dominates all three longest lines. However, if White plays **i2-c6**, two black stones will be captured at the e2-e8 line, but Black will take over domination at the b2-h5 line, threatening a white stone and a white Gipf.

When a four-in-a-row does not use spots at the edge of the board, it is often hard for the opponent to prevent this with one stone. A good planned attack in the middle of a blocked row often leaves the player an advantage.

This strategy somewhat overlaps with the centre-stones strategy. When the stones are centred, they occupy spots in the longest lines of the board.

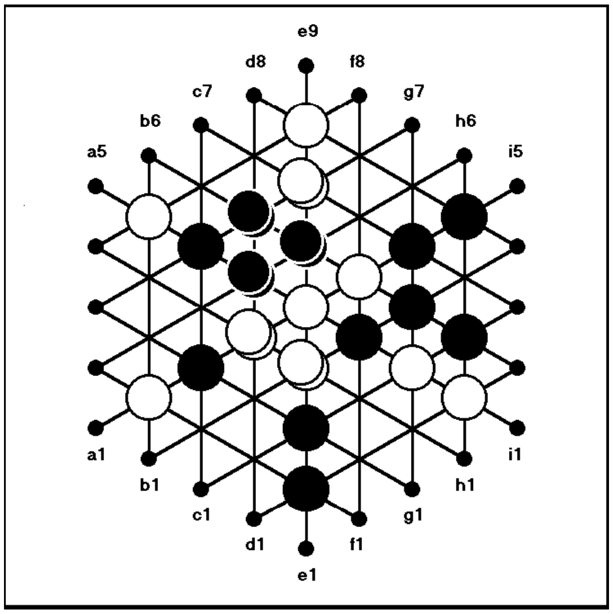


Figure 2.4: Example of the 'dominating lines' strategy

These are the lines which are tried to be dominated. When a player has many stones centred and the opponent has little centred stones, the player automatically dominates the longest lines.

2.2.4 Exchange stones

As in most board games, when a player has a material advantage, meaning that he has lost fewer stones than his opponent, it is useful to exchange stones. Sometimes a player is in a position that he can choose either to defend a stone from his own, or to capture a stone from his opponent, leaving the opponent to capture the players stone. When the player has more stones in total than the opponent he should usually trade, but when he has fewer, he should defend his stones. This is because when a player has an advantage, he wants to get the game over as soon as possible, so he does not run the risk of making a mistake and loosing the advantage. Especially if played with a clock, the player who has a better position can run out of time and loose. The shorter the game lasts, the less risk there is to run out of time.

2.2.5 Conclusion

The strategies described are always useful for human players, who implicitly compare the pro's and cons of a strategy in a particular situation and make a good combination of strategies.

It is harder to let a computer do these comparisons in each particular situation. Some strategies overlap, making the effect too strong, or contradict, neutralizing both strategies. Therefore these strategies have all to be tested apart from each other, as well as in combination with each other. In section 6.5 the results of the implementation of some of these these strategies are displayed.

Chapter 3

Complexity analysis

In this chapter the complexity of the game Gipf is examined. The analysis is split up into the state-space complexity and the game-tree complexity. The complexity of the game is important for the decision between a knowledge-based approach and a brute-force approach. Besides it gives an idea whether the game is solvable [3].

3.1 State-space complexity

The Gipf board is small compared to other games like draughts or chess, only 37 fields can be occupied by stones. One should expect a relatively small state-space complexity.

A first upper bound for the state-space complexity can be derived as follows: each spot on the board can be occupied by a white basic stone, a white Gipf stone, a black basic stone, a black Gipf stone or can be left empty. These are five possibilities on 37 places. Next to board are the stones in reserve for both players. This can be seen as two extra spots, each can be occupied in 19 different ways (from eighteen to zero stones). The player to move can be seen as the third additional 'spot', with two ways to occupy it. The upper bound is $5^{37} \times 19^2 \times 2 = 5.3 \times 10^{28}$.

To get a more accurate upper bound, one has to consider the fact that each party has only 18 stones, of which he can only make 9 Gipf stones. Each Gipf stone leaves two stones less to put on the board as basic stones.

At first there is the choice to put white Gipf stones at any position on the board, so there are $\binom{37}{\textit{number of white Gipfstones}}$ combinations possible with *number of white Gipf stones* minimal 1 and maximal 9. This leaves (18 - 2 $\times$ *number of white Gipf stones*) white basic stones to put on the board at (37 - *number of white Gipf stones*) positions. The same calculation applies to the black stones, only there are just (37 - (*number of white Gipf stones* + *number of white stones*)) empty places left at the board for the black Gipf stones and (37 - (*number of white Gipf stones* + *number of white stones* +

number of black Gipf stones)) choices for the black basic stones.

Next to these configurations of stones at the board, the number of stones in reserve for both the white and the black player must be taken into account. The number of stones in reserve for white is maximal the total number of stones (18) minus twice the number of white Gipf stones at the board minus the number of white stones at the board. The minimal number of stones in reserve is zero, although this implies that this position is the last one in a game. The same counts for the black stones in reserve.

The stones in reserve can be seen as two different playing spots, both have the maximum number of stones according to the position plus one (for zero stones) different ways to be occupied.

Which player is to move can be seen as a third additional spot. This 'spot' can have the values White and Black, so two possibilities.

All combinations of numbers of different stones on the board combined with stones in reserve and the player to move provide the total number of possible board configurations. When all this has been worked out, the next equation is reached:

$$2 \times \sum_{W_g=1}^{9} \sum_{W_b=0}^{18-2W_g} \sum_{B_g=1}^{9} \sum_{B_b=0}^{18-2W_g} \binom{37}{W_g}\binom{37-W_g}{W_b}\binom{37-W_g-W_b}{B_g} \binom{37-W_g-W_b-B_g}{B_b}(18-2W_g-W_b+1)(18-2B_g-B_b+1) \quad (3.1)$$

where W_g is the number of white Gipf stones, W_b is the number of white basic stones, B_g the number of black Gipf stones and B_b the number of black basic stones.

When (3.1) is fully written out and simplified, the following equation is left:

$$2 \times \sum_{W_g=1}^{9} \sum_{W_b=0}^{18-2W_g} \sum_{B_g=1}^{9} \sum_{B_b=0}^{18-2W_g} \frac{(19-2W_g-W_b)(19-2B_g-B_b)\times 37!}{W_g!W_b!B_g!B_b!\times(37-W_g-W_b-B_g-B_b)!} \quad (3.2)$$

The solution of this equation is $7.9223 \times 10^{24} \approx 10^{25}$.

This is still an upper bound, since there are some board configurations which will never occur in a game situation. Some examples of these positions have been given in figures 3.1 to 3.3.

In figure 3.1 a position has been given where both Black and White have two stones in reserve, all the others are at the board, all Gipf stones. A last move of one of the players would have put a stone at the edge of the board, or would immediately have been taken off in a four-in-a-row. There never could have been a last move, because each move puts a stone at the edge of the board, except when the stone forms part of a four-in-a-row and is taken off immediately. Since there are only two stones in reserve there could not have been a four-in-a-row in the last move (minimal two of the stones in the row should have been Gipf stones, there is no possibility to form a four-in-a-row using the Gipf stones at the board).

In figure 3.2 one of the two rows should have been taken off. Besides this, the rows could not have been created in the last move, because minimal one stone in the rows should have been connected to a spot at the edge of the board, the same reason as in figure 3.1.

In figure 3.3 rows should have been taken off many moves before this position originated. Many positions can be thought of where the same counts.

To summarize: a position where four-in-a-rows exist for both colors cannot exist, a position without stones at the edge of the board can only exist if it was possible to take stones off in the last move. More situations can be thought of that are less general.

The number of illegal boards seems rather small. Positions without stones at the edge of the board that are illegal are hard to find, there will not be a significant number of these positions. There are a lot more positions with two four-in-a-rows of different color, however it will be shown that this number of illegal positions will not influence the complexity.

There are 48 ways to put exactly four stones in a row at the board. If there is a white four-in-a-row at the board, the possibilities for another black four-in-a-row are between 20 and 40, depending on the position of the white row. This yields maximal $48 \times 40 = 1920$ different possibilities to put two four-in-a-rows at the board.

An upper bound for illegal boards because of minimal one four-in-a-row for both players can be derived with a equation like 3.1. At the board eight spots are occupied by the two rows, leaving 29 spots to choose from for the other stones. Minimal four stones for each player are used in the four-in-a-row, leaving maximal 7 Gipf stones or 14 basic stones. The equation is as follows, similar to equation 3.1:

$$1920 \times 2 \times \sum_{W_g=0}^{7} \sum_{W_b=0}^{14-2W_g} \sum_{B_g=0}^{7} \sum_{B_b=0}^{14-2W_g} \binom{29}{W_g}\binom{29-W_g}{W_b}\binom{29-W_g-W_b}{B_g}$$
$$\binom{29-W_g-W_b-B_g}{B_b}(14-2W_g-W_b+1)(14-2B_g-B_b+1) \quad (3.3)$$

The solution to equation 3.3 is 5.4×10^{22}. While this is an upper bound, the total complexity will be $7.9223 \times 10^{24} - 5.4 \times 10^{22} = 7.8683 \times 10^{24}$, no significant difference.

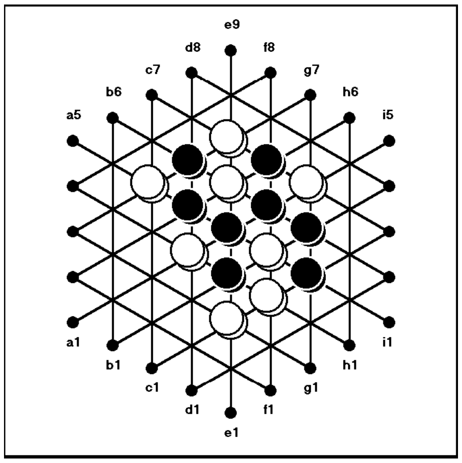


Figure 3.1: Position never could have been reached.

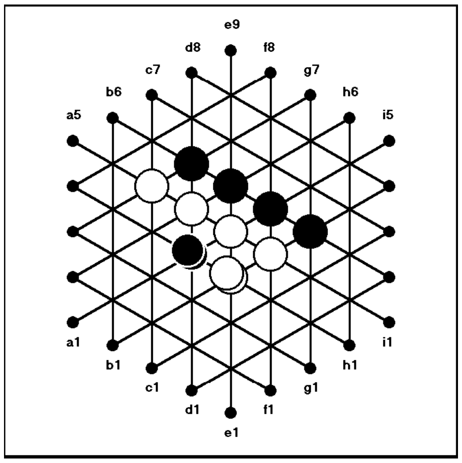


Figure 3.2: The two rows never could have been created here.

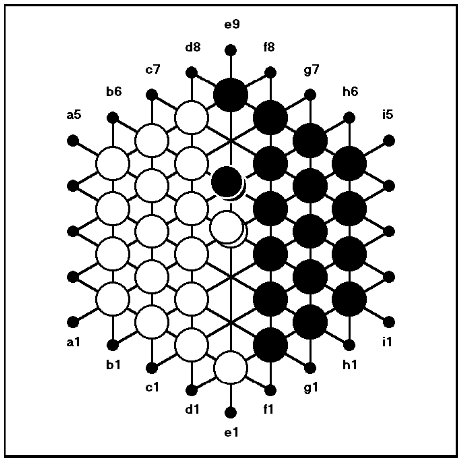


Figure 3.3: Many rows should have been taken off.

3.2 Game-tree complexity

The game-tree complexity is the number of different games that can be played. It correspondents with the number of leaf nodes of the game tree, with depth equal to the average number of plies in a game and a branching factor equal to the average number of possible moves in each part of the game.

The length of a game depends on the strength of the players. The discussion on the game length of Gipf, when played by the program GIPFTED can be found in section 6.2.2. When looking at the strongest games, searched six ply deep, this length is .., that is .. moves for each player. Looking at games played by human beings in tournaments that were not resigned early in the game, adapted from [40], the average length is 40.5 moves. At the match between GF1 and GIPFTED during the CMG 6th Computer Olympiad 2001, the average game length was 48.5 moves (games in [?]). Taking these three numbers into account, the average game length for the complexity calculation can be round to 45 moves, that is 90 ply.

The number of moves possible from a position varies. This number is influenced by some different aspects.

If a spot at the edge of the board is occupied, there are two different moves to enter that spot. If the spot is empty, the two moves can be seen as one move.

If all spots at a line at the board are occupied, the line is blocked, the two moves over that line are not possible.

When after a shove move four stones need to be taken off and there is a Gipf stone among them, the number of possible moves increases with one, namely the possibility to keep the stone at the board. When one is forced to take stones off at the beginning of his move and there is a Gipf stone involved, the number of possible moves doubles because before each shove move the choice has to be made to take the Gipf stone or leave it at the board.

Another situation can occur when a player has a choice to take off between two rows. When a Gipf stone is at the crossing of the two rows, the player has the choice to take either one of the rows together with the Gipf stone from the board or take both rows and chooses whether to take the Gipf stone or not. This situation leads to four different moves (if no other Gipf stones are in the rows to take off).

A game always starts for both players with playing a Gipf stone. After this move they may choose whether to put a Gipf stone or a basic stone at the board, until the moment that they have played the first basic stone. Looking at the games played at tournaments [40] most players play three or four Gipf stones before they play their first basic stones. It is expected that the branching factor during these first moves is twice the branching factor during the rest of the game. The fact that the board is (almost) empty in the first moves of the game lowers this branching factor to about 45.

The calculation of the average branching factor during the whole game when played by the program GIPFTED is thoroughly described in section 6.2.1. This average branching factor will be used to calculate the complexity of the game. The branching factor during the whole game is 29.3.

The calculation of the $_{10}log$ of the game tree complexity is as follows:

$$_{10}log(B^{2L}) = 2L \times_{10} log(B) \tag{3.4}$$

where B is the average branching factor during the whole game and L is the average number of moves played during a game. This leaves

$$90 \times_{10} log(29.3) = 132.0 \tag{3.5}$$

So the game tree complexity of Gipf is about 10^{132}.

3.3 Conclusion

Now that the two complexity numbers are known, it is interesting to compare these numbers to other games to get an idea of the complexity. Chess, for instance, has a state-space complexity of 50 and a game tree complexity of 123, 10 $\times$ 10 checkers has a state-space complexity of 30 and a game tree complexity of 54.

In [2] an overview of the complexity of games of the Olympic List has been given. In figure 3.4 the complexities of these games, together with Gipf, are presented.

It can be seen that Gipf has a relative high game-tree complexity and a relative low state-space complexity. According to [3] it is at the boundary between solvable and unsolvable games. If it is solvable it will be by brute-force methods.

Games with a big game tree and a small state space (like Amazons) can be approached best by brute-force methods, while games with a big state space and a small game tree (like Go-Moku) can be approached best by knowledge-based methods. From figure 3.4 it can be seen clearly that Gipf belongs to the second group of games. Therefore brute-force methods will be the best way to have a program play the game of Gipf.

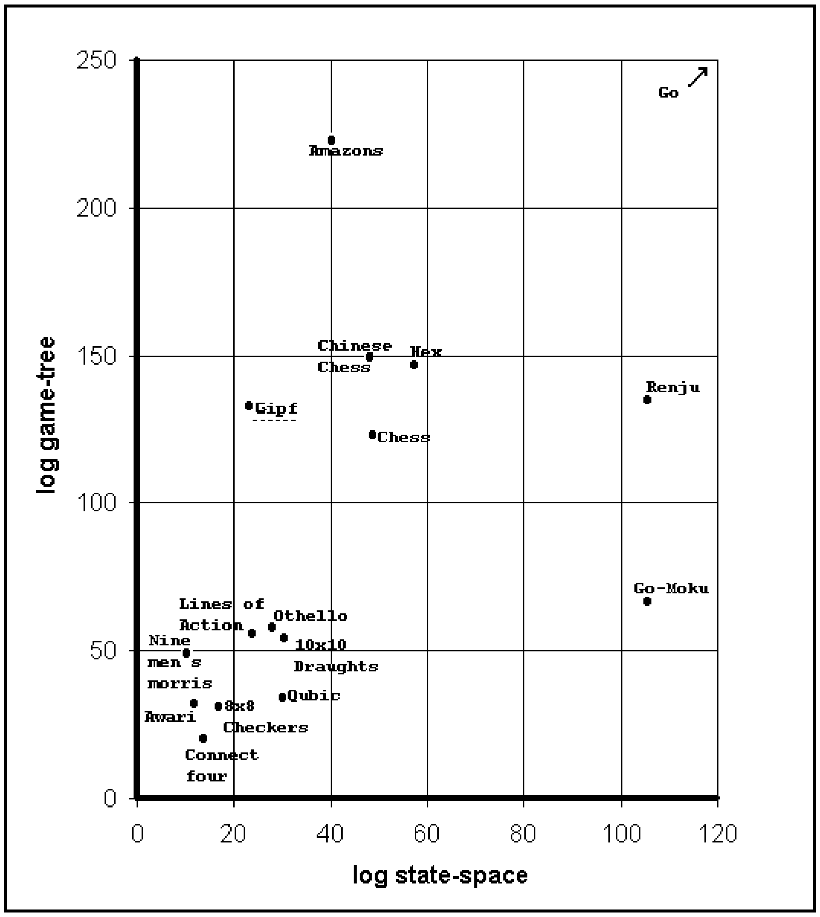


Figure 3.4: The complexity of different games

Chapter 4

Search

4.1 Introduction

The only way to let computers play board games is by having them search for a best move. The way they search and the amount of resources used by the search can vary for different approaches of the problem of playing a game.

As noted in section 3.3, the best way to approach Gipf is by brute-force techniques. This means that search is the main technique for obtaining the best move in a position.

There are different techniques to make the search more efficient. The search techniques used in the program are described below, followed by a short examination of techniques that may be useful for the game, although they have not yet been implemented.

4.2 Minimax search

To have a computer select the best move in a certain position, it has to 'think' ahead. This means that it has to apply all moves possible in that position one by one and return the best one. This leads to a search tree. Each level in the tree is a move further in the game. Such a level (half a move in a two-player game) is called a *ply*. The algorithm that selects the best move for a position using this search tree is called the *minimax* algorithm [24].

Minimax uses a *move generator* and an *evaluation function* to calculate the best move in a given position for the player to move. The move generator generates all possible moves for a given position, whereas the evaluation function calculates a value for it. It is assumed that there are two players, White and Black. Usually the evaluation value is high if White is in a good position and low if Black is in a good position.

From a given board, minimax generates a game tree existing of all positions that can be derived from the initial board by applying a finite number of moves. The root of the tree is the initial board. At the first level all positions follow the initial board by applying one move. At the second level all positions follow positions at the first level by applying one move for the opponent. This can go on until all positions are final states. The average number of moves possible from a board in this tree is called the *branching factor*.

In Gipf, there are 42 possibilities for the first ply in the game. For each of these possibilities there are 42 possible moves for the second ply. This leads to 1764 possible board states. At the third and fourth ply there are 84 possibilities, yielding 12,446,784 possible board states after only four plies. It is clear that it is not possible to construct the whole search tree because of memory and time overflow.

The minimax algorithm uses depth-first search instead of breadth-first. Levy and Newborn give some reasons for this [21]. First the amount of memory used by breadth-first search is much greater than when using depth-first search. With depth-first search only the branch that is being explored needs to be stored in memory, with breadth-first search the whole tree needs to be stored. Second the control strategy for depth-first search is simpler than for breadth-first search. It is always well defined which node to explore next, opposed to other techniques where a lot of jumping around in the tree will occur.

Each branch of the tree is searched until a fixed depth. When this depth is reached, the value of the board at that depth is evaluated and given back to the node one ply higher in the tree. From this node all other children are evaluated and the best score is given to its parent, one ply higher in the tree. If at the node White is to play, the child with the highest score is selected and the node is called a MAX node. If Black is to play, the lowest-valued child is chosen and the node is called a MIN node.

In this way all nodes in the tree obtain a value. The tree has to be traversed from left to right, yielding values to nodes from leafs to the root. An example is given in figure **??** for a three-ply-deep tree with branching factor two.

The number of nodes that has to be examined is the branching factor of the tree to the power of the search depth ($\textit{branching factor}^{\textit{depth}}$). Due to the exponentially growing number of nodes that have to be examined, several techniques are used to reduce this number.

4.3 α-β search

The main pruning technique used is called *α-β search*. This technique, already invented in 1958 by Newell, Simon and Shaw [26], was thoroughly examined by Knuth and Moore [19]. It is used in almost all board-game-playing computer programs. There are so far no practical alternatives to α-β, at least for computer chess according to Junghanns [18]. Gipf is in the same complexity range as chess, so it is obvious to use α-β pruning.

The α-β algorithm used in the application is an extension of the algorithm described in [28] pp. 132. It keeps track of an upper (β) and a lower bound (α) of the values in the search tree. If in a MAX node a value is obtained that is larger than β, it is assured that the MIN player can reach a node with a smaller value (the node with value β). The MIN player will always play for the smallest value, so the remainder of the subtree after this MAX node does not have to be examined, because it will never be chosen by a MIN node higher in the tree. An example is given in figure **??**. The same applies to a MAX node: if in a child (a MIN node) a value is obtained that is smaller than α, the rest of the subtree under the MAX node can be pruned.

From this it is clear that the higher the value of α and the lower the value of β, the more branches are pruned. To get the highest α and the lowest β, the best move for each player has to be examined first. However, if the best move is already known, there is no need to search anymore. Without knowing exactly the best move, it is possible to estimate which moves are the best. The nodes to be examined are ordered by these estimations.

The savings of using α-β search are very large, especially when a good node ordering is used. The minimax algorithm needs to explore b^d leaf nodes, where b is the branching factor and d is the depth of the search tree. When using α-β with a random ordering, the number of nodes to be explored is about $b^{3d/4}$ [28]. With a perfect ordering, the number of leaf nodes is only $b^{d/2}$ [37]. It turns out that with a good node ordering the result is very close to the number of leafs in a perfectly-ordered tree.

In the next sections the techniques used to order the nodes and to reduce the search tree in other ways are described.

4.4 Iterative deepening

Iterative deepening is a technique that does not seem to be obvious, intuitively. It has first been described in [32] and later rediscovered by many others, including [34] and [10]. Since then, all chess-playing programs have been using it.

With iterative deepening, instead of one search operation of a previously-determined depth, a sequence of increasingly-deeper searches is carried out, until a fixed depth is reached or the search has ran out of time. If, for example, one wants to search three ply deep, at first a one-ply-deep search operation is done, then a new two-ply-search operation, followed by a new three-ply-search operation. So the first ply is searched three times, the second ply two times and the third ply once.

This seems to cost a lot of extra time, but after each search operation, the new search is started with the *principle continuation*, or *principle variation*, of the former search. The principal continuation is the sequence of best moves found in a search operation (see 4.6.1). There is a large chance that these are also the best moves for a one-ply-deeper search, or at least very good moves. Because α-β prunes more if the search has started with a good move, the pruning effort will be more than the time it costs to do the searches over again.

Another important advantage of iterative deepening is when playing a timed game. Before this technique was used, it was very hard to estimate the depth of a search reachable for making a move. It could turn out that the search took one hour while there was only three minutes time or that a position was searched too shallow for the time left. Using iterative deepening, the shallower search can be used to estimate the time needed for a deeper search. When it turns out that the search takes too much time, there is always the result of the search one ply shallower, so the search can be stopped at any time.

4.5 Transposition tables

In a game like Gipf, where two players must move one stone by turns, it occurs that a different sequence of moves leads to the same position. Especially when the board has a low complexity this will often happen. When a position has already been explored, it seems unnecessary to explore it again. If the position is stored in a table, together with the relevant information that was retrieved by exploring it, it is possible to look up that information when the same position occurs again. Such a table is called a *transposition table* [11, 34].

It is not possible to store all board positions that are explored, because of memory restrictions. Therefore the positions are stored in a *hash table* of fixed size. Each position is converted to a sufficiently large number, called the *hash value*. Part of the hash value is used to map the position onto an entry in the table, using it as an index. The rest of the number is used to distinguish between positions that map onto the same entry. The hashing method that is used mostly is *Zobrist hashing* [39].

4.5.1 Zobrist hashing

Zobrist hashing is an easy method, which costs little calculation time and can even be done incrementally. For each different combination of spot and occupation a unique random number is generated, of the same size as the hash value. For Gipf there are 37 spots at the board and 5 possible occupations for each spot (empty, white stone, white Gipf stone, black stone, black Gipf stone). So $37 \times 5 = 185$ different random numbers are used.

To calculate the hash value, the XOR operator is applied to the binary random numbers for each spot at the board. If the hash value is large enough, this gives a unique number for each possible position.

Because in Gipf the same position can occur with a different number of stones in reserve, the number of stones in reserve for White and for Black can be seen as extra 'spots' at the board, with 18 possibilities each. The player to move is also taken into account to calculate the hash value. Therefore there is need for $185 + 2 \times 18 + 2 = 223$ different random values in total. To calculate the hash value, the XOR operator needs to be applied to 40 different numbers.

4.5.2 Replacement schemes

The size of the table might be much smaller than the total number of positions investigated for a move. Therefore different positions will be mapped onto the same entry in the table. To decide which position is stored in the table there is need for a *replacement scheme*. There are some different possibilities. The scheme DEEP replaces a value in the table if the new value is searched at least as deep as the old value. A better scheme is BIG [6]: the old value is replaced when there are more nodes searched to obtain the new value than there were to obtain the old value.

If a transposition table is sufficiently big, doubling the size of the table hardly increases its usefulness [6]. It is therefore better to use the memory not for a bigger table but for a two-level table. This means that at each entry, two positions can be stored. The replacement scheme to use is TWOBIG1 [7]. A new value is compared to the first stored value in the entry. If there are more nodes investigated to obtain the new value than there were to obtain the stored value, the first stored value is replaced with the new value (the same as BIG). If this is not true, the second stored value is replaced by the new one, without further investigation. As a result a new value is always stored, replacing the smallest one of the two entries in the table.

4.5.3 Errors

Because the transposition table is smaller than the number of possible positions, errors occur. There are two types of errors.

The first is called a *type-1 error*, or *hashing error* [21]. This occurs when two different positions have the same hash value. This is a serious error because two different positions map onto the same entry, while this is not recognized. The search process will continue with wrong information about a position. Usually there are more board positions than different hash values, so sometimes this error can occur. In some games it can be prevented by testing whether the transposition move is legal in the current position. In a game like chess this should be sufficient because there is little chance that the best move is legal in both positions. In a game like Gipf, most moves are usually legal (except for four-in-a-row moves), so this strategy will not work. When it occurs, there is not much to do about it, but in practice it can be prevented by taking a larger range of hash values.

The chance of having at least one error can be calculated with the following formula [35]:

$$P(at\,least\,one\,error) = 1 - e^{\frac{-M^2}{2N}} \tag{4.1}$$

Here M is the number of positions that are to be mapped onto the table and N is the number of different hash values possible ($2^{number\ of\ bits}$). Using this formula it is possible to calculate how large the range of hash values must be to eliminate the chance of type-1 errors.

The other type of error, the *type-2 error* or *collision*, is less serious. This error occurs when two different boards map onto the same table entry. The way to handle this type of error has been explained in 4.5.2.

4.5.4 Using transposition tables

Beside the hash value of the investigated position, there are minimally four other information fields important to store in the table. Depending on the replacement scheme, other information can be useful as well, but the folowing five items should be in each transposition table [22, 8].

key This is the second part of the hash value. The n least significant bits are used for the index of the table of size 2^n, the rest of the bits are used as a key, to distinguish different positions that map onto the same table entry. The key is also called a *lock*, since to recognize identical positions the key of a new position is compared with the lock of the previous position.

move The best move for the position is stored. When the position appears to be the best, this move needs to be carried out. Furthermore it is used for node ordering and in the principal continuation.

score The value of the position searched is stored as the score. This can be the real value, the α value (when the search was cut off by α) or the β value (when the search is cut off by β).

flag To distinguish between a bound (α or β value) and a real value, the flag is stored. This flag can have the value ‘real’, ‘upper bound’ or ‘lower bound’.

searchDepth This item indicates the depth to which the position is searched. To see if the value is reliable enough the search depth is needed. When using the replacement scheme DEEP, this item will also be used to decide which colliding position has to be stored or kept in the table.

Knowing all these information items, a choice has to be made what to do if a position is encountered that has been stored in the table. There are three possibilities:

- The flag indicates a real value and searchDepth is equal or bigger than the number of ply still to be searched in the tree from the considered position. The score can be used as a real score for the position, without any further search. The move can be used for the principal continuation.
- The flag indicates a lower or an upper bound and searchDepth is equal or bigger than the number of ply still to be searched in the tree from the considered position. The score is a bound and can be used like that. If the score is a lower bound, α might be adjusted according to the score. If the score is an upper bound, β might be adjusted according to the value of the score. If α is bigger than β after the adjustment, a cut off occurs and no further search is needed. If this is not the case, the move can be explored first in a further search because it was the best in the earlier search, so it can be used for node ordering.
- SearchDepth is smaller than the number of ply still to be searched in the tree from the considered position. The score is not reliable because the search process had been too shallow. Only the move is used for node ordering. It is first explored in the further search because, if it was the best move in a shallow search, there is a good chance that it will also be the best move in a deeper search.

From recent studies it appears that the contribution of the bound values is much larger than the contribution of the real values when using minimal window search (see section 4.7) [8]. This finding was opposite to the traditional idea behind the usage of transposition tables.

4.6 Node ordering

In most programs that rely on search, very much effort is done to order the moves in the tree properly. As stated in section 4.3 the savings in a tree that is nearly optimally ordered are very large. It will usually be more than 99% of the number of evaluated nodes.

Some ordering heuristics have already been described. The ordering from the iterative deepening (4.4) has only effect on the first ply of the tree. This is in fact the most important ordering in most programs. It is actually a special case of the killer heuristic, to be described in the second subsection.

The order of importance of the different techniques is the order in which the subsections are put. Here the ordering from the transposition table comes between the principal variation and the killer heuristic.

4.6.1 Principle Variation

When a best move is discovered, the value is based on a series of moves, as many as the search is deep. This series is called the *principal variation* (PV) and is the best path through the tree. The PV can be stored for each iteration in the search.

If a PV is the best path for an n-ply search, it is likely to be the best path for a (n+1)-ply search also. When searching the next iteration, at each ply, the search can start with the move for that ply from the PV of the former iteration.

The idea of principal variation is formalized when used with a *minimal window* [23]. This is further described in section 4.7.

When the program is searching in the time of the opponent, the PV is used. Because in the time of the opponent, the move he will make is unknown, the next move in the PV is the best move to start with.

4.6.2 Transposition move

In the last paragraph of section 4.5.4 the importance of the bound values in transposition tables has been stated. When a position has already been investigated, but the score stored in the transposition table is a bound, the best move for the position can be used for the node ordering. Also when score is a real value but the position has not been investigated deep enough, the best move for the position can be used.

Because each investigated position is stored in the transposition table, the contribution of transposition tables to the node ordering is high. Details about the transposition tables have been described in section 4.5.

4.6.3 Killer heuristic

The killer heuristic follows the same principle as the ordering from iterative deepening. It is based on the assumption that if in some position a move is best, the same move is likely to be best in another position at the same depth in the search. If another move now turns out to be best, the killer move is replaced by the new move.

Usually one killer is kept up for each ply in the tree. There have also been experiments of more killers at the same level in the tree and of using killers stored for some level also at other levels. An extensive description of killer moves can be found in [1].

4.6.4 History heuristic

The history heuristic is based on the fact that during the search, good moves appear in many principal variations. If a move is good in one branch of the search, there is a good chance that it is also good in another branch. Schaeffer [30] presented the idea of a table where all these moves are stored.

The history heuristic can be seen as a special case of the killer move. Instead of storing one ore two killers, every move in the tree until then is stored in a table, together with a score that indicates the frequency of being best. The moves to be explored in the tree are sorted by the score in the history table.

For chess a 64 x 64 table is commonly used to store all possible moves, i.e., indexed by 64 'from' squares and 64 'to' squares. Only the moves that are encountered in the tree get a score. Each time an interior node is left, the score of the best move from that node is increased by 2^d, where d is the depth of the subtree searched to obtain that best move.

For Gipf there is only need for 42 entries in the history table. This is because there is always a maximum of 42 different movements for a stone. The rest of the history heuristic can be implemented similarly as for chess.

4.6.5 Domain specific ordering

In most games there are different types of moves. In chess, for example, we can distinguish capture moves, checking moves, promotion moves, etcetera. Some of these moves generally are better than others. It is possible to order the move list initially, depending on the type of moves.

In Gipf there are essentially three types of moves: normal moves, moves where a player only takes his own stones of the board and capture moves. An addition to these three types of moves are the moves where a player takes one or more Gipf stones back and the moves where a player captures a Gipf stone. An ordering from good moves to bad moves could be:

1 Capturing Gipf stone

2 Capture move

3 Normal move

4 Take off own stones

5 Take off own Gipf stone

Of course there are many exceptions to this specific ordering. Therefore it can be overruled by all other ordering techniques.

4.7 Windowing techniques

In the α-β search, the interval between the values of α and β is referred to as the *window*. This window is originally set to $[-\infty, \infty]$. Here ∞ means a greater number than the value of a winning position.

The purpose of α-β search is to make the window narrower by adjusting α and β, so that more cut offs occur. Looking at this idea, it is not obvious to initialize the window to $[-\infty, \infty]$, if there is any information available about the magnitude of the returned value.

Several ideas have been developed to narrow the window, since the narrower the window, the more efficient the search. A drawback of this approach is that if the true value of the subtree falls outside the window, the subtree needs to be searched again with a wider window. The search is said to *fail low*, if the returned value is smaller than α, and to *fail high*, if the value is bigger than β.

One idea is to use the score of the former iteration as an estimation for the score of the next iteration [21]. When using this technique in chess, a window is used with a typical width of the value of two pawns, around this score. Say, the value of the former iteration was 5 and the value of a pawn is 10, the window would be $[-5, 15]$. If the search fails low, the window will widen to $[-\infty, 15]$, if it fails high, the window will become $[-5, \infty]$ for the re-search.

This idea is improved to *minimal window search*, first published by Pearl in his Scout algorithm [27]. It is generalized to *Principle Variation Search* (PVS) [23]. Instead of using the value of the former iteration for a window, the search starts with an infinite window. The PV of the former iteration is followed for the search from the first branch. At each ply, the search is started with the move in the PV at that level.

The window is then adjusted to $[v, v + 1]$, where v is the minimax value of the search so far. The rest of the branches are searched with this minimal window. If the value of a subtree is strict larger than v, this subtree needs to be re-searched with a window $[v, \infty]$. The value v will be enlarged to the value from this last search.

Usually, the first moves of the PV of the former iteration, will be the same as the first moves of the current PV. In this case, no re-search has to be done. If this is not true, it is important to have a good node ordering. If the minimal window fails at a branch, the best situation is when that branch leads to the best move. Otherwise more re-searches have to be done, which lead to an increase in search time. In case of a badly ordered tree, PVS will lead to an increase of search time, instead of a saving!

4.8 Quiescence search

All search techniques described so far are meant to do a search faster. The quiescence technique actually slows down a search. It is used to make a search more reliable and therefore make the program play stronger at a fixed search depth.

During a quiescence search only positions that are labelled ‘unquiet’ are taken into account. A position is called ‘unquiet’ when there is a big chance that the evaluation value of the board changes much after the next move. An unquiet position could follow a capture move or in chess a check move. All other positions are called ‘quiet’ or ‘dead’ positions.

When in a normal α-β search a leaf node has been evaluated, the position is labelled ‘quiet’ or ‘unquiet’. If it is labelled ‘unquiet’ an extensive search is done from that node, regarding only unquiet positions. Only when all positions are quiet, the search stops. The α-β value of the extensive search is used as an under bound for the evaluation value of the leaf node.

The idea is already proposed in the 1950s for a program playing checkers. After that, it is used for chess as well [11]. It has been used to prevent the *horizon effect* [5]. The horizon effect occurs when a search is done to a fixed depth. At the ‘horizon’ of the search (the maximum depth) the situation can occur that a single move can make a dramatic change to the value of the position. For example in chess, when the position at the leaf node has an unprotected queen for White that can be captured by Black in next move, this will not be noticed in the evaluation function. The position might therefore be evaluated as good for White, while it is really bad for White.

This situation can be prevented when using quiescence search. In the example, the evaluation node is a position where Black is to play, within the tree it would be a MIN node. In this node, the minimum of first the actual evaluation value of the position and second the quiescence value of all moves that lead to unquiet positions, is taken as the evaluation value. The quiescence values of all unquiet children of the leaf node are derived in exactly the same way: the maximum of the evaluation value of the node and the quiescence value of the unquiet children if it is a MAX node, otherwise the minimum. In games like chess where repetition of moves is not allowed, the quiescence search will always end in a quiet position. If a repetition can occur, a maximum quiescence search depth needs to be set.

The algorithm used in the program comes from [31]. Schrüfer also proposes three mechanisms to forward prune the quiescence search. At first, if the evaluation value of a MAX leaf node is bigger than β, there is no need for a quiescence search. At second, the evaluation value of a parent node can be used to adjust its child’s α value. At third, an optimistic bound for each child is derived before the child is actually expanded. The bound can be used to prune the quiescence search.

For the game of Gipf it seems good to use capture moves for the quiescence. It might also be useful to take all four-in-a-row moves into account, together with all threats on stones. A threat could be three stones of one color, threatening a stone of opposite color with the possibility of capturing it in one move.

4.9 Conclusion

The game of Gipf is a game with a small state space and a big game tree. A brute-force approach seems to be the best way to implement the game. Because Gipf has a complexity similar to the complexity of chess, α-β search with search enhancements have been investigated.

Most techniques have been discussed in this chapter, details about the implementation of these techniques can be found in chapter 5. The results of that implementation indicate whether the techniques were useful to the game of Gipf or not. These results have been described in chapter 6.

Chapter 5

Implementation

5.1 Representation of the board

The original Gipf board exists of seven vertical lines, numbered ‘b’ to ‘g’, and two virtual vertical lines where stones can be put on, ‘a’ and ‘h’. At each line, all dots and spots are numbered from one, below, to the number of dots and spots at the upper end of the board. This representation is very clear for people playing the game.

However, for calculations it is not the ideal representation. When a move is made from ‘a1’ to ‘b2’ and only the last spot in that row is empty, the move is made from ‘a1’ to ‘h5’. For the first four stones, the letter as well as the number is incremented by one. For the last three stones, only the letter is increased, the number remains ‘5’. This makes the move non-linear. Therefore the representation of the board in the program differs from the common representation for a Gipf game, making all moves linear.

At first: instead of letters and numbers, only numbers are used for both the horizontal and the vertical direction. Second: from vertical line ‘f’, the dots and spots are numbered from the upper to the lower end, starting with ‘9’. If a stone is moved from ‘a1’ to ‘h5’ it is said to go from ‘11’ to ‘88’. This leaves a vertical direction (the first number remains constant), a horizontal direction (the second number remains constant) and a diagonal direction (the first and the second number are increased by the same amount).

5.2 Construction of the search tree

As stated in section 4.2, the search is done in a search tree. This tree has to be constructed depth-first. The construction is done by the α-β algorithm, in the way described in section 4.3. The only parts of the construction that are interesting, are the move generation, the evaluation function and the function that detects final states.

5.2.1 The move generator

To select the best move, the program needs a list of all the possible moves from a given board configuration. It then can apply each different move to the board and evaluate the new position.

The generation of a move consists of three steps. At first the detection of a four-in-a-row caused by the last move, then the addition of all possible shove moves in the position and at last the detection of a four-in-a-row caused by the shoving of a stone.

Detection of a four-in-a-row at the beginning of the move

If the last move of the opponent causes a four-in-a-row for the one to move, the stones have to be taken off. If there are one or more Gipf stones in that row, there is the opportunity for each stone to take it off, or to keep it on the board.

To discover a four-in-a-row, the program considers the last move from the opponent. Only the stones which have changed position by this last move, can cause a four-in-a-row, otherwise this four-in-a-row had already existed before the last move. An exception for this is when four or more Gipf stones form a four-in-a-row. This situation can remain the same for several moves and is described in part 5.2.1. Only considering the stones that are moved, leads to a saving of time.

At first the direction of the last move is determined, being horizontal, vertical or diagonal. Then for each stone that was moved, the program looks at the other two directions for a four-in-a-row, adjacent to this stone. Finally, a four-in-a-row is looked for in the original direction, starting at the position where the last stone was added.

Sometimes, more than one four-in-a-row is found. In this case there are two possibilities: they have a common stone or they do not cross. If they don't cross, they both can easily be taken off. If they do cross, then there are some possibilities. If the stone at the crossing point is a basic stone, a choice has to be made which row to take off. Otherwise, if that stone is a Gipf stone, there is a choice to take off only one row, or take off both rows. Hereby the same thing for Gipf stones counts as noted before.

In case of only two rows crossing, all possibilities for taking those rows off are written out in the program, in order to let the move generation goes fast. If a row crosses with more than one other row or two rows cross with two other rows, which will happen very rare, a search process is started to find all the combinations to take off the rows. At first all possibilities to take off just one row are noted. Then for each possibility, the row is taken from the board and the original function to discover a four-in-a-row will be recursively called. In this case some possibilities will be looked at more than once. This can be seen as a disadvantage of this approach, it is very hard to filter out doubles. With the use of transposition tables the disadvantage is small. When a move is applied which has already been applied, there will be an entry in the transposition table so that the move does not need to be investigated any further.

Each of the possibilities and combination of possibilities mentioned here are the beginning of a different move. So every continuation of the move shall be applied to all the different beginnings that have already been attained.

Generation of stone moves

All the possibilities to shove a stone onto the board have to be examined. Originally there are 42 different moves, but a stone can only be shoved into a position if there is an empty spot somewhere on the row in the direction of the stone move. The 42 possibilities have been written out in the programming code, if the move is legal, it is added to each partial move (taking stones from the board) in the list of moves.

In the beginning of the game, it is possible to shove a Gipf stone on the board instead of a normal stone. Gipf stones can be played until the first basic stone has been played. So before the first basic stone has been played, there are twice as much possibilities for a move as there are after the first basic move.

Detection of a four-in-a-row after the stone move

After the new stone is shoved onto the board, a four-in-a-row can occur for the player who is at play. This row has to be taken off, the same way as in part 5.2.1. The last move which is passed on to the four-in-a-row function, is now not the move of the opponent, but the first part of the current move.

Detecting a four-in-a-row costs relatively much time. Therefore it is better not to search for one, if it is assured that no four-in-a-row can exist. A move can only cause a four-in-a-row if before the move, three stones of the same color are within four spots at a line. Instead of searching 42 times (for each possible stone move) for a four-in-a-row, a search for three stones at four spots is done, once. This search leaves a few moves that could possibly cause a four-in-a-row. Only these moves are examined further. This leaves a great saving in the time it costs to generate moves.

Four Gipf stones in a row

When the situation occurs that four or more Gipf stones form a row, there is a problem with the normal detection of four-in-a-rows. The normal detection originates with the fact that a four-in-a-row is always caused by the last stone moved onto the board. A four-in-a-row of only Gipf stones can remain at the board for several moves. To overcome this problem a special procedure is written to detect a four-in-a-row of Gipf stones.

When a move is done and a four-in-a-row of only Gipf stones is detected (in the normal way), this fact is remembered. A procedure searches for all possibilities to take off this row, before it looks at other four-in-a-rows. When the row of Gipf stones is broken, there will be no searching for it anymore.

With the occurrence of a four-in-a-row of only Gipf stones, another problem arises. In this situation the number of possible moves becomes very big. In the worst case, having seven Gipf stones in a row, the number of possible moves will go up to more than 75000. This is too much to go through in the search process.

There are two solutions implemented for this problem. At first, if a four-in-a-row occurs of only Gipf stones, there is very little chance that taking off all Gipf stones is a good move. Usually four Gipf stones in a row is a very desirable situation. Therefore the assumption is made that it is never good to take more than two Gipf stones from the board. This only counts for the stones in the color of the four-in-a-row!

The second is a makeshift solution. If during the move generation the list grows over 10000 moves, the move generation is stopped and the search will return the best move already achieved in the former iteration.

These two solutions prevent the search process to overflow and use all the time for one move. Although the first solution is not very common in the game world, usually all legal moves are considered, it is necessary to solve this problem in Gipf and the first one is a better solution than the second one.

5.2.2 Final board states

When searching for the best move, the program needs to know when it has lost or won. If a move leads to a win in some plies, it has to play it and if a move leads to a loss, the program will play another move. When the final state is reached, the program needs to recognize it and stop the game with a message who has won. A function has been written to recognize an final state.

There are two ways to loose the game, run out of stones or run out of Gipf stones. The function returns 'white won' when Black has no stones in its reserve and is to move or if there are no black Gipf stones at the board. It returns 'black wins' if the former counts for White.

Say the white player is at play and has two stones left in his reserve. He thinks four plie ahead. The white player will not recognize that, if he does not retrieve stones in this move or the next, he has lost. This is because technically, White has only lost when he is at play and he has no stones in his reserve. Only after five ply, he will recognize it.

After the second move (three ply ahead), however, White can be sure of his loss, except for Black doing a stupid move, forcing White to take stones back. Assuming that the opponent is a reasonable player, the program determines that a player has lost if he has no stones left in reserve, after his move. This leads to a recognition of a lost position two ply higher in the tree, so it can usually be prevented a move earlier.

5.3 The evaluation function

The evaluation function determines the strength of a position for both the white and the black player. It returns a integer value which is a positive value if the position is better for White than for Black and a negative value if Black has a stronger position. The value of the function is constructed of a basic value for the number of stones for each player. Some positive or negative values are added to the basic value, according to some strategies described in section 2.2.

The numbers and equations given in this section are for calculating the white values. The values for Black are the same, except for a minus sign in front of each number or equation.

The values of the different stones in the game are calculated as follows: each basic stone at the board has a value of 230, each Gipf stone has a value of 465. The value of each stone in reserve increases when there are less stones in reserve, according to the following formula:

$$\textit{value of reserve} \quad = \quad n \times (280 - \lceil \sqrt[3]{n-1} \times 20 \rceil) \tag{5.1}$$

where n is the number of stones in reserve.

5.3.1 Centre stones

By adding something to the value of stones that are more to the centre of the board, the heuristic of centre stones is applied. The following equations are used for each stone:

$$\textit{extra value basic stone} = (4 - Max(|x-5|, |y-5|, |x-y|)) \times 10 \tag{5.2}$$

if the stone is a basic stone and

$$\textit{extra value Gipf stone} = (4 - Max(|x-5|, |y-5|, |x-y|)) \times 20 \tag{5.3}$$

if the stone is a Gipf stone. In these equations x is the letter of the position of the stone, converted to a number, and y is the number of the position of the stone, both converted according to the representation described in section 5.1.

For example, a Gipf stone at position d5 will get a basic value of 465 and a centre value of $(4 - Max(|4-5|, |5-5|, |4-5|) \times 20 = (4-1) \times 20 = 60$, applying equation 5.3, that is a total value of $465 + 60 = 525$.

If the centre stones heuristic is not used, each basic stone will have 20 added to its value and each Gipf stone will have 40 added to its value.

5.3.2 Cluster Gipf stones

In the program GIPFTED the clustering heuristic is only applied to Gipf stones. If a Gipf stone is close to another Gipf stone, something is added to the value of the stone. Only the closest Gipf stone is taken into account. The added value is derived as follows:

$$\textit{extra value Gipf stone } i = \lceil (6 - Min\{d_{ij} | i \neq j\})^3 / 5 \rceil \tag{5.4}$$

where d_{ij} is the shortest distance between Gipf stone i and Gipf stone j, measured along the shortest path from i to j. The minimal distance is 1 and the maximal distance is 6.

When using the example in the former paragraph, say there is another Gipf stone at e6 and one at e4. The distance between d5 and e6 is 1 and the distance between d5 and e4 is 2. The extra value for the Gipf stone at d5 is $\lceil (6-1)^3/5 \rceil = 25$. The total value will be $525 + 25 = 550$.

5.3.3 Exchange stones

When a player has captured more stones than his opponent, each captured stone of both players adds a little to his evaluation value. The more stones are captured, the more is added to the evaluation value according to the next formula:

$$\textit{extra value} = (c_b - c_w) \times (c_w + c_b) \times 10 \tag{5.5}$$

where c_w is the number of captured white stones and c_b is the number of captured black stones.

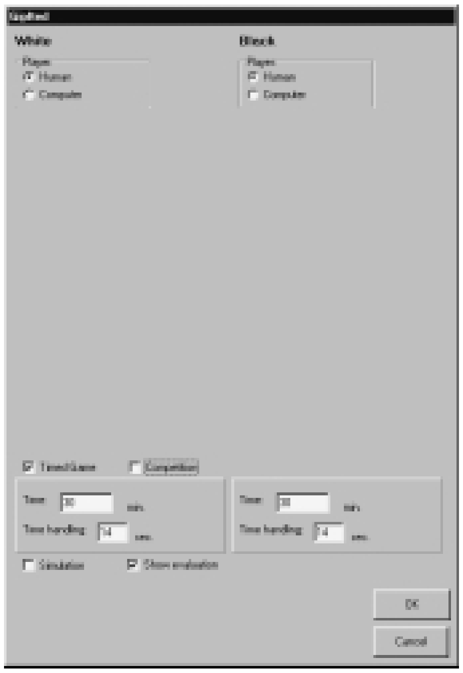
The formula holds for both the white and the black player, since a positive evaluation value indicates a good position for the white player and a negative evaluation value indicates a good position for the black player.

The total evaluation value of a board is build up out of these values, the value for the white player minus the value for the black player. To prevent the program playing the same game over and over again, a small random value is added (usually in the range of $[-10, 10]$).

5.4 The user interface

The user interface of the program GIPFTED is aimed at the testing of the different features implemented in the program. When selecting a new game from the menu, a form is presented where all features can be set for both players. This form is shown in figure 5.1. Here both players are set to human players.

In figure 5.2 the form is set to do a simulation. Both players are played by the computer, all search and evaluation features are used, except for the


Gipfted

White
Player
[✓ Human]
Computer

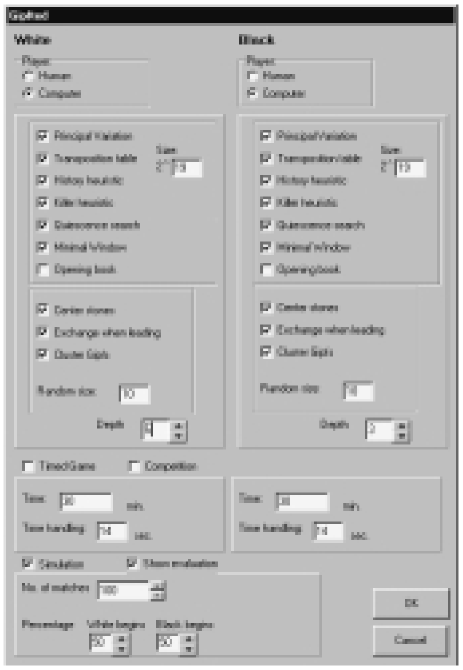
Black
Player
[✓ Human]
Computer

[✓ Timed Game] Competition

Time: 30 min. Time handling: 14 sec.
Time: 30 min. Time handling: 14 sec.

Simulation [✓ Show evaluation]

Figure 5.1: New game for two players


Gipfted

White
Player
Human
[✓ Computer]

[✓ Principal Variation]
[✓ Transposition table] Size 2^ 19
[✓ History heuristic]
[✓ Killer heuristic]
[✓ Quiescence search]
[✓ Minimal Window]
Opening book

[✓ Center stones]
[✓ Exchange when leading]
[✓ Cluster Gipfs]
Random size: 10
Depth: 4

Black
Player
Human
[✓ Computer]

[✓ Principal Variation]
[✓ Transposition table] Size 2^ 19
[✓ History heuristic]
[✓ Killer heuristic]
[✓ Quiescence search]
[✓ Minimal Window]
Opening book

[✓ Center stones]
[✓ Exchange when leading]
[✓ Cluster Gipfs]
Random size: 10
Depth: 2

Timed Game Competition

Time: 30 min. Time handling: 14 sec.
Time: 30 min. Time handling: 14 sec.

[✓ Simulation] [✓ Show evaluation]

No. of matches 100
Percentage White begins 50 Black begins 50

Figure 5.2: New simulation

opening book. The number of simulations is set to 100, 50% of which is started by White, 50% by Black.

When 'New game' is chosen from the 'Game' menu, the game start with the initial game board, that is, an empty board and 18 stones for each player. It is also possible to do a board setup and to play from a non-starting position. To get the setup screen, 'Setup board' has to be chosen from the 'Board' menu. The screen is shown in figure 5.3. At the left on the screen a choice can be made what type of stone to place at the board. By clicking spots at the board, a stone is placed. At the right the number of lost stones can be chosen, optionally the last move can be given in the 'best move' field. The last move is needed if a four-in-a-row needs to be recognized before a move is made.

A board setup can be saved with the 'Save board' option in the 'Board' menu. A name for the board can be given in a dialog that pops up. Also during a playing situation a board can be saved in the same way. With the 'Load board' option, a saved board can be loaded into the setup board mode.

When the button 'Play' is clicked, the same form will be shown as when 'New game' is chosen. After the players and options are selected, a new game starts from the given board position.

The setup board can also be used to add positions to the opening book. The position from where the opening move has to be done can be set up at

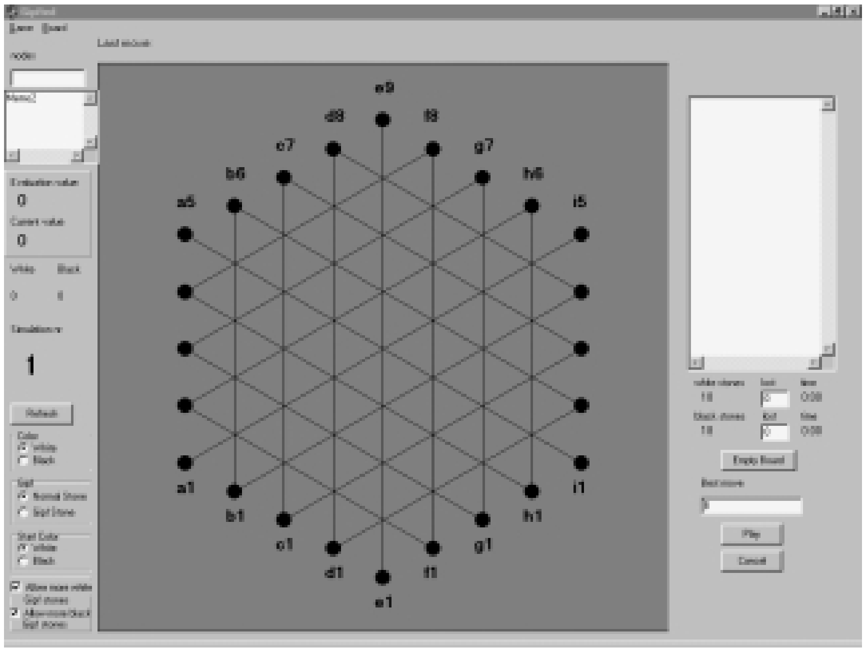
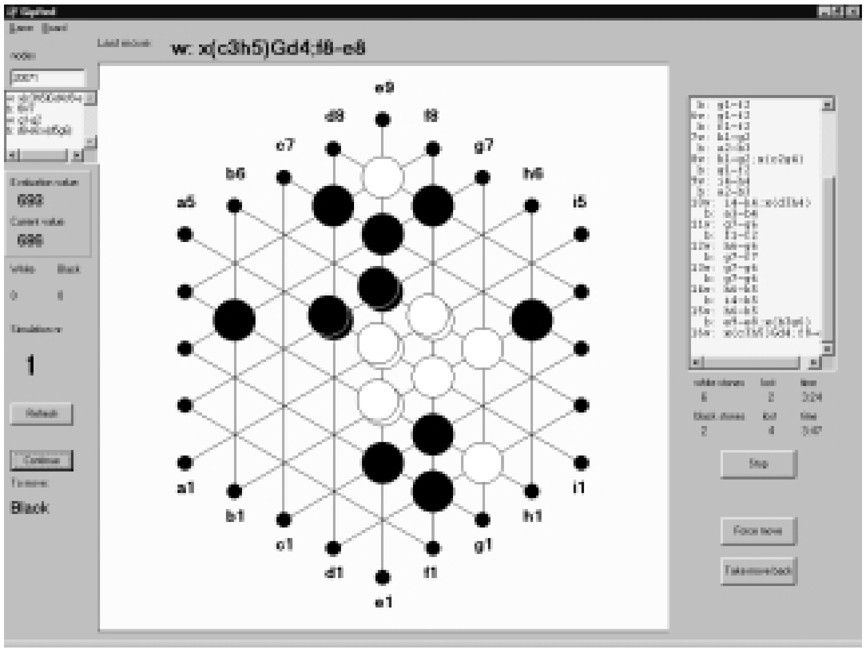


Figure 5.3: Setup board



w: x(c3h5)Gd4;f8-e8

```
 b:  g1-f2
6w:  g1-f2
 b:  i1-f2
7w:  h1-g2
 b:  a2-b3
8w:  h1-g2;x(c2g4)
 b:  g1-f2
9w:  i3-h4
 b:  a2-b3
10w:  i4-h4;x(d3h4)
  b:  d1-d4
11w:  g7-g6
  b:  i1-f2
12w:  h6-g6
  b:  g7-f7
13w:  g7-g6
  b:  g7-g6
14w:  h6-h5
  b:  a4-b5
15w:  h6-h5
  b:  e9-e8;x(h3e6)
16w:  x(c3h5)Gd4;f8-
```

Figure 5.4: Halfway a game

the setup screen. In the 'best move' field, the best move from the position has to be given. When 'Add opening' is chosen in the 'Board' menu, an opening book has to be selected and the position with the move is saved in that opening book.

When an actual game is played it looks like figure 5.4. At the left from top to bottom the following items are shown:

- The number of nodes investigated by the last move of the computer player.
- The expected best path found by the computer player in the last iteration, if PVS is used.
- The evaluation value that is expected by the player who did the last move is shown (if that player is the computer, otherwise the value is 0).
- The evaluation value of the current position.
- The number of games won by White and by Black so far in a simulation.
- Number of the current game the simulation.
- Refresh button. If this button is pushed, the board is refreshed at the screen.
- Pause button. In figure 5.4 the game is actually paused. It continues when this button is pushed. When the game is in progress, pushing this button will pause it.
- The play color who is to move.

On top of the play board the last move done is shown. At the right of the board from top to bottom the following items are shown:

- In the memo all moves done so far are shown.
- The number of stones in reserve for Black and White, along with the number of stones that are captured by the opponent. The third item is the time played for each player during this game or, when a timed game is played, the time left for each player.
- The stop button to stop a game or a simulation.
- When the 'Force move' button is pushed, the computer player will immediately do a move, using its last completed iteration.
- The 'Take back' button is used to undo the last move.

Figure 5.4 shows an advantage for White. At the current position the evaluation value is 686, White expects a value of 693 in four moves, not a real change. White expects Black to do f8-f7 in the next move.

When a human player plays the computer or another human being, he has to do moves. He can put a stone at the board by clicking one of the black dots. After he clicks a legal position to move the stone to, the move is definitive and the opponent is to move. Before he completed his move, the player is allowed to change it.

If a choice has to be made in taking Gipf stones or rows of the board, the human player will be asked which one to take. A dialog appears in the left lower corner and the game will continue after the player has answered the question in the dialog. Along with the dialog, the stones that are asked about are given a red border for the sake of clearness.

Chapter 6

Experimental results

6.1 Introduction

In this chapter the experimental results of the program GIPTED are given. At first, the different properties of Gipf have been examined. At second, the results for the different search techniques used in the program are given. The experiments on these techniques consist of sessions of minimal 100 simulations. With a confidence level of 95%, the confidence interval of the mean number of nodes is within 5% of that mean. The chapter concludes with results on the used evaluation heuristics.

6.2 Properties of Gipf

6.2.1 Branching factor

The branching factor of the game will be used to calculate the game-tree complexity. The game tree of which the complexity is calculated, exists of all different games that can be played. To create this game tree, if possible, a minimax search should be done from the empty board, searching until the longest possible game has been finished. It is obvious that practically this is not possible for most games, but this complexity gives a good estimation for the search difficulty of the game.

The average branching factor of a game can be approached in different ways. At first, the number of moves possible from all game positions that occur in a number of games can be taken into account. The second approach is to take the mean of the number of moves possible from all board positions encountered in the game tree. The average number of moves will be different for both approaches.

When the branching factor in the whole tree is used, many positions are encountered that will not occur in a reasonable game. Therefore to get a reliable estimation for the number of possible moves one has to choose from when playing the game, the first approach has to be chosen. However, to get the best approach of the complexity it is better to use the branching factor through the whole tree. An α-β tree looks more like the minimax game tree than a series of playing boards does.

The branching factor differs a little for different search depth's and for the search enhancements used. Especially the use of transposition tables decreases the average branching factor a little.

The branching factor has been measured in all experiments on the search enhancements. Because the differences in branching factor are small between the experiments, it seems sufficient to give the average. The average branching factor is 29.3.

6.2.2 Average game length

6.2.3 Starting advantage

The game of Gipf has never been investigated thoroughly. It is therefore unknown whether starting the game is an advantage, a disadvantage or neither.

For all games that have been played by the program GIPFTED to investigate the different search-properties in section 6.4, it has been noted who started the game and who won. Only games played for the investigation of search techniques were taken into account, because these techniques do not influence the game itself, only number of nodes investigated and the time needed to do this may change. All games are searched to a fixed length and use quiescence search.

The results are presented in table 6.1. For each search depth the total number of games won by the beginning party are given, divided by the total number of games played. Further the starting advantage is stated as a percentage of the number of games won without an starting advantage.

Depth	*Total # games*	*Games won by the starting party*	*Starting advantage*
2	1223	610	-0.2%
3	1220	674	10.5%
4	1601	920	14.9%
5	1301	681	4.7%
6	x	x	x%

Table 6.1: Results of the starting advantage

6.3 Number of Gipf stones

6.4 Search techniques

6.4.1 Minimax vs. α-β search

The difference between minimax and α-β searchα-β search is very big, even without any node-ordering techniques or transposition tables. From ply .. on, α-β search reduces the search with more than .. %. This reduction is expected, according to previous research [19, 26, 28].

6.4.2 Transposition tables

Gipf is a game with a relative small state-space complexity (see section 3.1). Transposition tables are therefore expected to reduce much in the search tree, because a position will often be encountered more than once.

The main advantage however of using transposition tables comes from the better node ordering by using the bound values in the table [8]. This might explain why using the transposition table the gain is smaller than for instance when using the killer heuristic instead (see table 6.6). When the search is done without the transposition table, the other ordering techniques will most likely find the best node to explore first.

	With transposition tables		**Without transposition tables**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	218	0.4 s	0.9%
3	1322	2.2 s	1439	1.2 s	8.1%
4	5252	1.9 s	6627	2.8 s	20.7%
5	30398	14.9 s	47021	18.2 s	35.4%
6	140879				

Table 6.2: Results of the use of transposition tables

From table 6.2 it can be seen that the reduction is quite good, but it would be interesting to see what reduction the transposition table as the only search enhancement would gain. These results are presented in table **??**.

6.4.3 PVS minimal window

The PVS minimal window technique can be bad for the search. When the PV often changes in the next iteration, many subtrees have to be re-searched. In table 6.3 it can be seen that the search without PVS if faster than the search with PVS for all tested search depths except the three-ply and the five-ply search.

The search uses the PV from the former iteration. The good result at the five-ply search is derived from the four-ply iteration. From the results it seems that the PV is better estimated by an even number of ply than when searching an odd number of ply. To test this an extra experiment has been done: a seven-ply search with and without PVS minimal window.

	With PVS minimal window		**Without PVS minimal window**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	174	0.4 s	-24.1%
3	1322	2.2 s	1323	1.6 s	0.1%
4	5252	1.9 s	4803	2.2 s	-9.3%
5	30398	14.9 s	33074	11.3 s	8.1%
6	140879	x s	120465	54.5 s	-16.9%

Table 6.3: Results of the use of PVS together with transposition tables

	With PVS minimal window		**Without PVS minimal window**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	218	0.4 s	185	0.4 s	-17.8%
3	1439	1.2 s	1610	1.9 s	10.6%
4	6627	2.8 s	8350	3.7 s	20.6%
5	47021	18.2 s	70628	25.2 s	33.4%
6	x	x s	351696	x s	x%

Table 6.4: Results of the use of PVS without transposition tables

6.4.4 Killer heuristic

From table 6.5 it appears that the killer heuristic is a very useful node-ordering technique for the game of Gipf. Up till five ply, it reduces almost as much as transposition tables. When looking at the results for the killer heuristic without the use of transposition tables (table 6.6), the reduction in nodes is even bigger. The saving of using the killer heuristic is much bigger than the saving of the transposition tables (see table 6.2). There might be an explanation why killer moves work so well. When there are some stones at the board, there are often many possibilities to create a four-in-a-row. In many cases when it is actually created, the whole board changes. In a lot

	With Killer heuristic		**Without Killer heuristic**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	256	0.4 s	15.6%
3	1322	2.2 s	1545	2.0 s	14.4%
4	5252	1.9 s	6430	2.6 s	18.3%
5	30398	14.9 s	43339	26.9 s	29.9%
6	140879	x	197885	86.0 s	28.8%

Table 6.5: Results of the use of the Killer heuristic together with transposition tables

	With Killer heuristic		**Without Killer heuristic**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	218	0.4 s	261	0.6 s	16.5%
3	1439	1.2 s	1860	2.7 s	22.6%
4	6627	2.8 s	12008	4.6 s	44.8%
5	47021	18.2 s	95890	x	51.0%
6					

Table 6.6: Results of the use of the Killer heuristic without transposition tables

of lines that were blocked, the block will be broken. If both players are of the same strength, the white and black stones are equally divided in most blocked lines. If one of the players could add one stone to the line, he will capture a lot of the opponent's stones. Often both players have to prevent creating a four-in-a-row.

Because of this fact, players often have little moves to choose from. In these cases it might be independent from the opponents last move which moves are the best. The killer move would be the best move for all positions at this depth in the tree.

	With History heuristic		**Without History heuristic**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	236	0.4 s	8.5%
3	1322	2.2 s	1425	1.5 s	7.2%
4	5252	1.9 s	6645	5.3 s	21.0%
5	30398	14.9 s	37562	12.7 s	19.1%
6	140879	x	171426	53.4 s	17.8%

Table 6.7: Results of the use of the history heuristic together with transposition tables

	With History heuristic		**Without History heuristic**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	218	0.4 s	245	0.3 s	11.0%
3	1439	1.2 s	1514	1.6 s	5.0%
4	6627	2.8 s	9111	2.6 s	27.3%
5	47021	18.2 s	59601	16.0 s	21.1%
6					

Table 6.8: Results of the use of the history heuristic without transposition tables

6.4.5 History heuristic

6.4.6 Domain-specific ordering

6.4.7 Quiescence search

6.5 Evaluation heuristics

6.5.1 Centre stones

6.5.2 Cluster Gipf stones

6.5.3 Exchange stones

	With domain-specific ordering		**Without domain-specific ordering**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	226	0.4 s	4.4%
3	1322	2.2 s	1435	1.7 s	7.9%
4	5252	1.9 s	6855	9.3 s	23.4%
5	30398	14.9 s	37833	12.5 s	19.7%
6	140879	x s	179046	x s	21.3%

Table 6.9: Results of the use of domain-specific ordering together with transposition tables

	With domain specific ordering		**Without domain specific ordering**		**Gain**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	218	0.4 s	231	0.5 s	5.6%
3	1439	1.2 s	1663	2.3 s	13.5%
4	6627	2.8 s	9182	12.5 s	27.8%
5	47021	18.2 s	65656	x	28.4%
6					

Table 6.10: Results of the use of domain-specific ordering without transposition tables

	With quiescence search		**Without quiescence search**		**Overhead**
Depth	*Nodes*	*Time*	*Nodes*	*Time*	%
1					
2	216	0.4 s	187	0.4 s	13.4%
3	1322	2.2 s	1188	1.5 s	10.1%
4	5252	1.9 s	5837	2.3 s	-11.1%
5	30398	14.9 s	31267	10.1 s	-2.9%
6	140879	x	139825	60.1 s	0.7%

Table 6.11: Overhead of the use of quiescence search

	All features on	
Depth	*Nodes*	*Time*
1		
2	216	0.1 s
3	1322	0.5 s
4	5252	1.9 s
5	30398	11.0 s

Table 6.12: Results of using all search and evaluation heuristics

Chapter 7

Conclusions and future research

Appendix A

The Gipf Tournament at the 6^{th} Computer Olympiad

A.1 Introduction

Gipf is a new game, invented by Kris Burm in 1996. It has been played by a group of human players since. At the CMG 6th Computer Olympiad 2001 the first computer match between two Gipf programs has been held. Gipf is being played at a hexagonal board, the two players alternately moving a stone from their reserve onto the board. By creating a row of four adjacent stones, a player can capture stones from his opponent and/or retrieve own stones to his reserve. A game is won when the opponent has run out of stones in his reserve or when the opponent has lost all of his Gipf stones. The latter are special stones that can be moved onto the board in the beginning of the game. The game and the notation used below is thoroughly explained at the web site *http://www.gipf.com/*.

A.2 The programs

Two programs competed at the Olympiad: GF1 written by Kurt van den Branden and Gipfted written by Diederik Wentink. GF1 ran on an AMD 1.3 GHz processor. It has a very fast move generation because it divides a move in three separate steps, only checking the board for rows to take of when it really uses the move. This approach makes move ordering considerably more difficult, so GF1 has to consider a lot of nodes. The main concept in the evaluation is the clustering of all stones. Gipfted ran on a Pentium III 833 MHz processor. It uses about 60 Mb memory for transposition tables. More ordering techniques are used than in GF1, but it has a slower move generation. Gipfted uses a small opening book. The evaluation uses a clustering of only Gipf stones. All in all GF1 was a bit faster than Gipfted; it searched about a half to one ply deeper on a total of six to seven ply per move.

A.3 The competition

In most of the games, both programs were at a par after the opening phase (around move 15). A win then of a single stone by one of the programs usually led to some more captures and the victory in 40 to 50 moves. GF1 has beaten Gipfted with six games to two (see table A.1). The four games that were most interesting are presented below. The (unannotated) scores of all games can be found at URL
http://www.cs.unimaas.nl/Olympiad/results/gipf.

Game	Player 1	Player 2	Score
1	GF1	Gipfted	1-0
2	Gipfted	GF1	1-0
3	GF1	Gipfted	1-0
4	Gipfted	GF1	0-1
5	Gipfted	GF1	0-1
6	GF1	Gipfted	1-0
7	Gipfted	GF1	1-0
8	GF1	Gipfted	1-0

Table A.1: Results of the Gipf tournament.

Game 1 GF1 - Gipfted GF1 started the match very well. Although the first stones are only captured after move 16, GF1 manages to get a very strong position from the beginning on. This leads to a pretty hopeless position for Gipfted, which faces defeat from move 15 onwards.
1. Gh3 Gc6 2. Gi2-g4 Gc7-c5 3. Gi2-f5 Gc7-c4 4. h4 b5 5. i3-g5 i4-f4 6. i3-f6 g6 7. b6-d6 b6-e6 8. i3-e7 f7 9. f2 a5-d5;x(c4-g6) 10. a4-d7 a4-e8 11. d8-d4 e9-e5;x(c5-f7) 12. x(e6-h3);b6-d6 a5-c5 13. h5 i4-g6 14. i4-f7 g7-g4 15. i3-f6 h6-h2 16. a5-g3 i3-f3 (see figure A.1) White has only one stone left in his reserve, so he has to retrieve stones. At this point, White has many choices both to retrieve stones and to capture a black stone. The white stones are positioned well, while the black stones are spread around the board. Since Black is threatening the Gipf stone at f3, White plays **17. i4-e4;x(f2-f7)Gf3** followed by Black's **17. ... d8-d3** to save the Gipf stone at d4. **18. i4-f7;x(d4-h5) c7-c3 19. c7-e7 c7-f6 20. d8-d4 d8-h5;x(e8-h5) 21. i2-e6 c2 22. i4-f4;x(c6-h3) x(c2-f4)Gd3;f8-f7** Now Black faces a loss of four stones against zero for White: the game is as good as won by White. **23. i4-g4 g7-d6 24. x(d4-d7);i1-f4 i3-g5 25. b6-c6 f8-f5 26. b6-d6 f8-f3;x(f3-f7) 27. i1-f4 i2-g2 28. x(g2-g5);c2 f7 29. b4 b3 30. g6 b6-b2 31. a4-d7 a1-d4**

32. a3-d3 a1-e5 33. e8 a4-f3;x(b2-e5) 34. c7-f6 h5 35. d8-d5;x(b3-g6) h3;x(h2-h5) 36. e9-e3;x(e3-e8)Ge4 g7-e7 37. c1-c3 d8-d5 38. x(b5-f4);b3 f8-f6 39. c1-c4 f8-f5 40. f2 b6-e6 41. d2 1-0

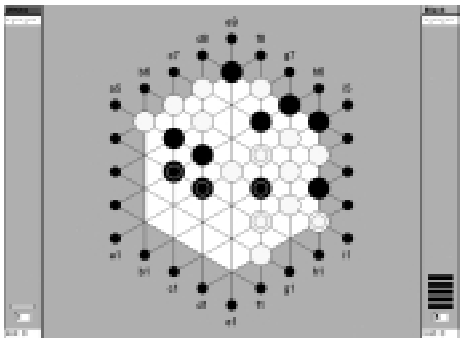


Figure A.1: game 1, after 16. ... i3-f3.

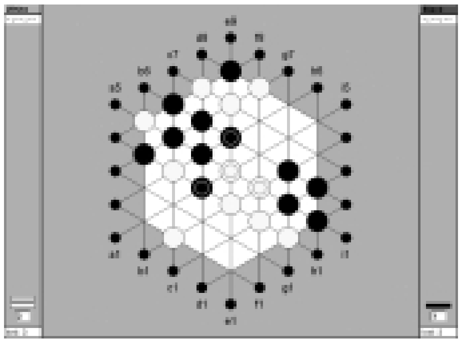


Figure A.2: game 2, after 23. h1-e4.

Game 2 Gipfted - GF1 The game develops equally. The early exchange of a Gipf stone (in move 12) does not seem to harm the play of the programs. After move 25, White has a slight advantage of one stone, which appears to be enough to win this game.
1. Gd2 Gc6 2. Gd1-d3 Gc7-c5 3. Gd1-d4 Gb5 4. b6-d6 c1-e3 5. c7-c4 a4-d7 6. c1-f3 c1-g3 7. c1-h3 d1-d5 8. g2 c7-c3 9. c7-c2 a5-e5 10. a4-e8 f2 11. g1-g4 b6-e6 12. b1-e4;x(c4-g2)Gc4 i2-f5;x(c6-h3)Gg4 13. h2 h1-h3 14. b1-e4 e9-e7 15. d1-d6 g1-b3;x(b3-f2) 16. d1-d3;x(d2-d7) h1-h4 17. i4-g4 i3-f3 18. g2 a3-b4 19. f8-d7 f8-c6 20. a5-f4 h1-d4 21. x(e4-h4);a3-d6 a3-f7 22. g1-g4 c7-c4 At this point, Black has only one stone left and three possibilities to retrieve stones. White plays **23. h1-e4** (see figure A.2), attacking the Gipf stone at d4. Instead of playing i2-f5, which seems the safest solution, Black plays **23. ... e9-e3;x(b4-e7)**, capturing one stone from White, but sacrificing a Gipf stone. From here on, White has an advantage and tries to exchange stones as much as possible. **24. e2 i1-c5** White takes his own Gipf stone for unclear reasons: **25. x(e2-e6);h1-b4;x(d4-g2)Gd4Gc4 c7-c4 26. c1-c3 b3 27. a2-e6 i2-f5;x(e6-h3) 28. d8-d6 e9-e7 29. b6-f5 a2-f6 30. e9-e4;x(b5-e8) x(c5-h2);a3-d3 31. e2 c1-c5 32. a3-e3 d1-f2 33. e1-e5 d1-g2 34. c6 b1-f4 35. f1-d2 d1-h2 36. h1-f3 a2-d5;x(b3-f2) 37. c7-c3;x(c2-c6) f7 38. i2-f2 d7 39. i1-g3x(d5-h2) b6-c6;x(c6-f5) 40. f1-f4 d1-h2 41. b4 d8-d6 42. a3-c5 b6-c6 43. d8-d5 c7-c4;x(b4-f7) 44. i1-g3;x(d5-h2) g1-e3 45. b1-b2 f1-f4 46. a1-c3 f1-f7 47. c7-e7 c1-g3;x(d2-g3) 48. a1-d4;x(b2-f5) f1-f3 49. h1-d4 e1-e3 50. h1-b4 g7-d6 51. a3-c5;x(b4-f7) a4-b5 52. b2 1-0**

Game 4 Gipfted - GF1 This game also starts equally. In the beginning White loses a stone, but Black has to take a Gipf stone of the board. After White is forced to take two of his Gipf stones of the board, Black is able to force White into losing sacrifices and wins easily.

1. Gd2 Gh4 2. Gc1-e3 Gi3-g5 3. Gd1-d3 Gh5 4. e2 i5-f5 5. e1-e4 i4-g4 6. h3 d1-d4 7. i2-e6 c6 8. h6-h2 b6-d6;x(c6-h3)Ge6 9. h6-h3 d1-d5 10. i4-g4 i4-g6 11. i3-f6 i1-g3 12. x(g3-g6);i4-g4 i5-g5 13. i4-g6 i4-f7 14. i3-e7 c1-f3 15. i4-f4;x(d3-h4) f8-f4;x(f3-f7) 16. e1-e5 c1-f3 17. f1-c2 i1-g3 18. h1-h4 e1-e6 19. h6-f6;x(d5-g6)Gd5 i1-d5;x(h2-h5) 20. x(d2-g3);f1-d2 f1-b2 21. c1-e3 h4 22. e8 g6 23. f7 h5 24. h6-h3 h6-h2 25. i5-f5 f8-d7 26. b1-b3 f8-c6 27. a1-c3 i3-g3;x(d5-h2) 28. d1-d3 c1-f3 29. b1-f4 With **29. ... c1-g3** (see figure A.3) Black forces White to take the row c6-f5. Along with this row White has to remove his second Gipf stone. **30. b6-d6;x(c6-f5)Ge6 e1-e6** White has to sacrifice a stone to prevent Black to capture two white stones. **31. x(b2-e5);e9-e5 x(e2-e8);h6-f6 32. h6-d5** Instead of taking the stone at g6, Black decides to attack White's last Gipf stone. **32. ... i4-e8 33. g7-g4 i4-e4 34. x(g3-g6);c1-e3 x(e3-e6);f8-f5 35. i4-g6 h6-h2 36. f8-f2 d1-d6 37. i2-g2 g1-g3** A series of sacrifices is the only thing White can do to postpone Black's victory for some moves. **38. i3-e3 x(d2-h3);g1-g3 39. d8-d2 x(d5-h2);i5-g5 0-1**

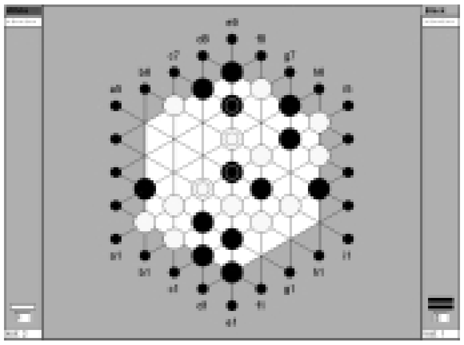


Figure A.3: game 4, after 29. ... c1-g3.

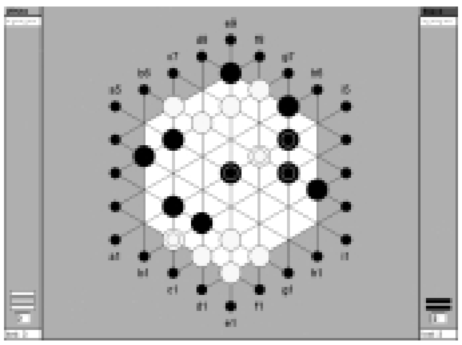


Figure A.4: game 8, after 25. g7-b4.

Game 8 GF1 - Gipfted The game starts different than the others, with White playing only two Gipf stones. This, however, does not harm White's play. Not much happens until round move 25, when White sets a double trap for both Black's Gipf stones. In the defence of his Gipf stones, Black loses many stones and loses the game.
1. Ge8 Gf7 2. Gf8-f6 Gh4 3. e9-e7 White only plays two Gipf stones! **3. ... Gi3-g5 4. f8-f5 i3-d7 5. d8-d6 f8-c6 6. f8-f4 f2 7. g1-e3 c7-c5 8. x(c5-f7);c7-c5 g1-d3 9. g1-c3 f7 10. g1-b3 h5 11. f1-f3 g7-b4**

12. d8-d5 i4-g4 13. b6-e6 c7-c4 14. x(b3-f6);f1-f6 x(d7-h4);h4 15. g7-e7 x(e7-h4);c7-c2 16. b6-e6 a4-d4 17. g1-b3 i5-e5;x(c3-h5) 18. b1-e4;x(c2-g4) h3 19. c7-c3 a3-d3 20. d2 a2-g6 21. c7-c2 a4-d4 22. e2 b6-g4;x(b3-g6) 23. c7-c4 a3-e7 24. e1-e4;x(b4-f3) g7-g5 25. g7-b4 (see figure A.4) So far, both players have lost three stones. White has a double threat on Black: both the stone at b4 and the Gipf stone at g4 are under attack. If Black plays i2-e6, White will respond with f8-f6 followed by f8-f4 capturing two black stones. Black sacrifices his stone at b4. **25. ... c7-c4 26. x(b4-f7);b6-d6 g7-g3;x(g3-g6)Gg3 27. e1-e4 f1-b2 28. g1-b3 i2-e6 29. c1-c5;x(b3-f2) e9-e7 30. b1-d3 e9-e3;x(c6-h3)Gf5** Black takes a Gipf stone back because otherwise it would be captured after **31. b1-f4 c1-c3**. From here White gets the upper hand. Black has to defend his last Gipf stone and loses more and more stones. **32. c1-c6 b1-g4 33. e9-e6 b5 34. d1-f2;x(e2-e8) a5-d5 35. b6-b4 a4-d4;x(c2-c6) 36. a2-c2 c1-c3 37. x(b2-e5);h2 h1-h3 38. i3-g3 f1-f3 39. h1-h4;x(d5-h2) i2-f5 40. c1-e3 f1-f4;x(f2-f5) 41. c1-f3 a4-c4 42. d1-d4 i2-e6;x(e6-h3) 43. b1-e4 c1-c5 44. x(b4-f3);e2 b1-e4 45. c1-c6 a4-d7 46. b6-b4 a4-e8 47. f1-b2;x(b2-e2) b6-d6 48. i3-g5 1-0**

Bibliography

[1] S. Akl and M. Newborn. The principle continuation and the killer heuristic. In *ACM Annual Conference*, pages 466–473. ACM, Seattle, 1977.

[2] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Universiteit Maastricht, Maastricht, 1994.

[3] L.V. Allis, H.J van den Herik, and I.S. Herschberg. Which games will survive? In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2: The second computer olympiad*, pages 232–243. Ellis Horwood, Chicester, UK, 1991.

[4] L.V. Allis, H.J. van den Herik, and M.P.H. Huntjens. Go-Moku solved by new search techniques. *Computational Intelligence: An International Journal*, 12(1):7–24, 1995.

[5] H.J. Berliner. *Chess as Problem Solving: The Development of a Tactics Analyzer*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1974.

[6] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes for transposition tables. *ICCA Journal*, 17(4):183–193, 1994.

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

[8] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Information in transposition tables. In H.J. van den Herik and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 199–212. Universiteit Maastricht, Department of Computer Science, 1997.

[9] M. Euwe. Mengentheoretische Betrachtungen über das Schachspiel. In *Proceedings*, volume 32, pages 633–642, Koninklijke Akademie van Wetenschappen te Amsterdam, 1929.

[10] J. Gillogly. *Performance Analysis of the Technology Chess Program*. PhD thesis, Department of Computer Science, Carnegie Mellon University, 1978.

[11] R.D. Greenblatt, D.E. Eastlake, and S.D. Crocker. The Greenblatt chess program. In *Proceedings of the AFIPS Fall Joint Computer Conference 31*, pages 801–810, 1967.

[12] A.D. de Groot. *Het Denken van de Schaker*. PhD thesis, Universiteit van Amsterdam, 1946. (in Dutch). An extended translation has appeared under the title *Thought and Choice in Chess*, Mouton, The Hague 1965.

[13] K. Handscomb. Game Review: Gipf. *Abstract Games*, 1, 2000.

[14] K. Handscomb. Game Review: Project Gipf. *Abstract Games*, 4, 2000.

[15] K. Handscomb. Interview with Kris Burm. *Abstract Games*, 6, 2001.

[16] H.J. van den Herik. *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. PhD thesis, Technical University Delft, 1983. (in Dutch).

[17] H.J. van den Herik, J.W.H.M. Uiterwijk, and J. van Rijswijck. Games solved: Now and in the furture. Technical report, Universiteit Maastricht, IKAT/Department of Computer Science, Maastricht, the Netherlands, September 2001.

[18] A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, 1998.

[19] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[20] D. König. Über eine Schluweise aus dem Endlichen ins Unendliche. *Mitteilungen der Universität Szeged, Tom. Fasc.*, 2–3:121–130, 1927.

[21] D. Levy and M. Newborn. *How Computers Play Chess*. Computer Science Press, W.H. Freeman and Company, New York, 1991.

[22] T.A. Marsland. A review of game-tree pruning. *ICCA Journal*, 9(1):3–19, 1986.

[23] T.A. Marsland and M.S. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–552, 1982.

[24] J. von Neumann. Zur Theorie der Gesellschaftspiel. *Mathematishe Annalen*, 100:295–320, 1928.

[25] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944. Second Edition 1947.

[26] A. Newell, J.C. Shaw, and H.A. Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–335, 1958.

[27] J. Pearl. Asymptotic properties of minimax trees and game-searching procedures. *Artificial Intelligence*, 14(2):113–138, 1980.

[28] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice-Hall, Inc., 1995.

[29] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM journal of research and development*, 3:211–229, 1959. Reprinted in E.A. Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 71–105. McGraw-Hill, 1963.

[30] J. Schaeffer. The history heuristic. *ICCA Journal*, 6(3):16–20, 1983.

[31] G. Schrüfer. A strategic quiescence search. *ICCA Journal*, 12(1):3–9, 1989.

[32] J.J. Scott. A chess-playing program. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 255–265. Edinburgh University Press, 1969.

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

[34] D. Slate and L. Atkin. CHESS 4.5 – the northwestern university chess program. In P. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer, New York, 1977.

[35] J.W.H.M. Uiterwijk. Déjà vu. *Computerschaak*, 15(2):84–91, 1995. (in Dutch).

[36] J.W.H.M. Uiterwijk, L.V. Allis, and H.J. van den Herik. A knowledge-based approach to Connect-Four. The game is solved! In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence: the First Computer Olympiad*, pages 113–133. Ellis Horwood, Chicester, UK, 1989.

[37] P.H. Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, second edition, 1983.

[38] E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In E. Hobson and A. Love, editors, *Proceedings of the 5th International Congress of Mathematicians*, volume 2, pages 501–504. Cambridge University Press, 1913.

[39] A.L. Zobrist. A new hashing method with application for game playing. *Technical Report 88, Computer Sciences Department, The University of Wisconsin, Madison*, April 1970. Republished in ICCA Journal, Vol 13, No. 2, pp. 69–73.

[40] www.gipf.com, the gipf homepage.