On Solving Pentago

Betrachtung der Lösbarkeit des Spiels Pentago

Bachelor-Thesis von Niklas Büscher aus Münster

Mai 2011

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Knowledge Engineering Group

On Solving Pentago
Betrachtung der Lösbarkeit des Spiels Pentago

Vorgelegte Bachelor-Thesis von Niklas Büscher aus Münster

Gutachten: Prof. Dr. Johannes Fürnkranz

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 21. April 2011

(Niklas Büscher)

Abstract

This thesis deals with the different ways to solve games, with a special focus on solving the game of Pentago. Pentago is a new and hence until now unvisited, zero-sum two-player game with perfect information. We approximate the *state space* and *game-tree* complexity of Pentago to draw assumptions on Pentagos solvability. We further introduce several algorithms which have been successfully used to solve other board games so far. Testing their applicability on Pentago shows, that none of them will succeed under reasonable resource constraints. To obtain more details on the game-tree we developed an Artificial Intelligence (AI) to play Pentago called *PentagoAI*. In comparison with other available AIs, *PentagoAI* reveals to be one of the actual strongest players for Pentago. Finally we can only make a guess about the game theoretic value of Pentago.

Zusammenfassung

In dieser Arbeit haben wir uns der Lösbarkeit von Pentago auseinandergesetzt. Pentago ist ein neues Nullsummen Brettspiel für zwei Spieler mit perfekten Informationen. Neben einer ausführlichen Betrachtung der Suchraum- und Spielbaumkomplexität von Pentago, diskutieren wir mehrere Algorithmen die in der Vergangeheit erfolgreich verwendet wurden um Spiele zu lösen. Die Vor- und Nachteile dieser Algorithmen werden vorgestell und es wird der benötigte Rechenaufwand für eine Anwendung auf Pentago geschätzt. Wir zeigen, dass keiner der vorgestellen Algorithmen Pentago mit sinnvollen Zeitaufwand lösen kann, solange wir nicht mehr Informationen über den Spielbaum und damit Lösungsbaum erlangen. Daher stellen wir eine Künstliche Intelligenz (KI) für Pentago vor, die auf einer Alpha-Beta Suche basiert. Im Vergleich mit anderen frei erhältlichen KIs zeigt sich, dass die von uns entwickelte KI *PentagoAI* eine der zurzeit spielstärksten KI Implementierung für Pentago ist. Über den spieltheoretischen Wert von Pentago können wir schlussendlich nur Vermutungen anstellen.

Contents

1 Introduction

This thesis deals with the different ways to solve games, with a special focus on solving the game of Pentago. Pentago is a new and hence until now unvisited, zero-sum two-player game with perfect information. Before going any deeper into the topic of solving, we start with an introduction into Pentago and its rules in Chapter 2. Chapter 3 follows with a short historical summary of the work done in the field of solving games so far. After examining the complexity of Pentago in Chapter 4, Chapter 5 introduces several algorithms to solve games and points out their applicability on Pentago. We also discuss the reasons, why we are not able to solve Pentago. Chapter 6 gives a detailed insight in the program PentagoAI, written to play Pentago. Finally Chapter 7 concludes this work on Pentago.

The interested reader should be on the level of a graduate student in computer science. He should especially be familiar with the topics of estimating computational costs (Landau notation), data structures (hash-functions, trees) and he should have basic knowledge in the field of Artificial Intelligence (game-tree, minimax search algorithm). For simplicity reasons “he” will be the pronoun of choice in this thesis.
For readers who are familiar with the topic of solving games, it might be reasonable to skip the Chapter 3 and to skim the introductions of the described solving methods in Chapter 5. Readers who already played Pentago may skip smoothly the description of Pentago in Chapter 2. For all others it is suggested to read the chapters in order of their occurrence beginning with the introduction into Pentago in the following chapter.

2 Pentago

Pentago is an abstract strategy game for two players, invented by Tomas and Michael Flodén in Sweden, 2003. The Swedish company Mindtwister published the game in 2005 and has the rights of developing and commercializing the product [44]. Since its release, Pentago won multiple prices, examples are “Game of the Year 2005”, Sweden, “Game of the Year 2006”, France, “Spiel Gut”, Germany and “Mensa Select 2006”. A german championship was played in 2010[24].
Moreover Pentago is a zero-sum game with perfect information and very similiar to Gomoku (Connect5) on a 6x6 board, extended by a single element. Each placement of a marble is followed by a rotation of a 3x3 part of the board by 90 degrees. This game extension offers a much more complex game than a classic n-m-k-in-a-row game.

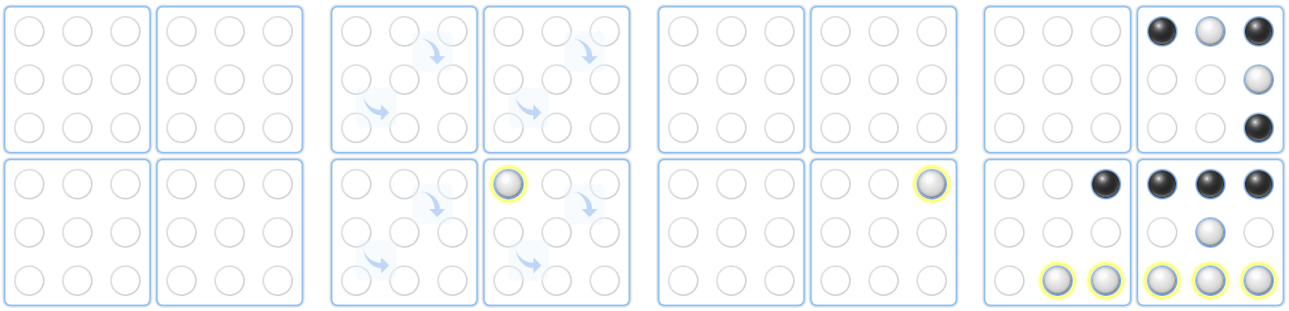
Figure 2.1: Official Pentago box layout by Mindtwisting games.

Explanations

Zero-sum games are games in which the win of one player is the loss of the other player and vice versa. Cooperative games are usually not zero-sum games.
Perfect information games are games where both player can see the whole gamestate. The game *poker* does not offer the whole gamestate to all players, because all players have access to their own cards and the board, but they do not have any valid information about the cards of their opponents. Therefore *poker* is a game with imperfect information.

2.1 Rules

The game Pentago is played on a 6x6 board, which is divided in four 3x3 sub-boards (further mentioned as quadrants). Figure 2.2(a) shows the initial setup of the board. Two players take part by turns in Pentago. Each move of one player consists of placing a marble of his own color onto an empty field on the board and turning one of the quadrants by 90 degrees, either clockwise oder anti-clockwise, illustrated in Figure 2.2(b) and Figure 2.2(c). No restriction of rotating based on previous moves exists. The game finishes with a win for the player, who gets five in a row at first 2.2(d). Five in a row can occur vertical, horizontal or diagonal, also before the sub-board rotation takes place. If no player achieves five in a row, the game ends after the 36th move with a draw. When both players get five in a row as a player turns the board, the game is also a draw.



(a) Initial board setup. (b) First marble set, Player has to rotate quadrant. (c) Position after first a clockwise rotation of the south-east quadrant. (d) Game ends, White wins the game.

Figure 2.2: Pentago Board in different game situations.

2.2 Related work

There are rarely few publications about Pentago. Schiffel [37] and Finnsson [15] take a short look at Pentago as a game in General Game Playing, moreover Schiffel uses Pentago as an example for automatically detecting symmetry features in board games. But as far as we know, no work on expert knowledge like transcriptions of tournament games or even opening-books has been done. However at least one comparable AI implementation exists, which is taken into account in Chapter 6.
The next Chapter 3 gives an introduction into solving games and its historical development, followed by a precise definition of solving.

3 Solving games and task definition

This thesis tries to figure out, whether and how Pentago is solvable within a reasonable amount of time. Before defining the task and discussing what is meant with reasonable amount of time we take a short look back in the history of Artificial Intelligence and explain the actual common sense of *solving* a game.

3.1 Historical background and motivation of solving games

Intelligent games are a research subject in the field of Artificial Intelligence for many years now. Beginning in 1959 Arthur Samuel wrote the first *checkers* Artificial Intelligence (AI), called ”checkers“ to play against human opponents [35]. Within a tournament in 1963 against a human checkers expert, the checkers AI won a single game, based on oversight of the human expert. The media stated this as “Checkers is solved”, even the AIs strength was probably lower than a human amateur and *checkers* was by far not solved.
But it required almost another 50 years to solve checkers by Jonathan Schaeffer and his group [30]. Inbetween these 50 years the Artificial Intelligence research on games developed many algorithms, strategies and ideas to solve games and also the development of the underlaying hardware exploded. So there must be a reason that it lasted almost 50 years to solve checkers:
At first we have to mention that *checkers* is probably the game with the largest complexity which has been solved so far. Second the gap in resource requirements between playing on or below amateur level and solving a game may be really large. Thinking of *chess*, where actual AIs are already able to beat grand-masters on consumer hardware, the full solution of *chess* with it enormous large game-tree is nowadays unfeasible. In conclusion the complexity of the games themselves seems to be the determining factor whether games are solvable. A detailed introduction into complexity of games is given in Chapter 4.
But where does the motivation to examine games comes from? Herik et al. [43] described, the interest of Artificial Intelligence researchers in strong game-playing programs as an important goal for more than half a century now. ”The principal aim is to witness the ”intelligence“ of computers. A second aim has been to establish the game-theoretic value of a game, i.e., the outcome when all participants play optimally.“ Heule et al. [19] added the motivational question: ”Can artificial intelligence outperfom the human masters in the game? “. Yet it is important to mention, that the methods to show intelligence and outperform human master can differ substantially from those for solving. But all together, games seem to be an interesting site to show Artificial Intelligence. Moreover solving games applies the Artificial Intelligence methods to larger problems which may become computational puzzles. Even so the topic of solving may be seen as a toy application, it is perfect to show that large knowledge based problems are solvable. The following selection shows a few nontrivial games, which have been solved so far and which have been used as a source for examining Pentago:

1980 Qubic: Qubic is a four-in-a-row in a three dimensional space (4x4x4) and was solved by Patashnik in 1980 [26] and later again by Allis [3, 4]. Patashnik used a combination of mathematics and computational power, wheres Allis developed a new search algorithm called Proof-Number search. Qubic is interesting, since it game-tree size (10^{34}) and state-space complexity (10^{20}) [19] is rather large. But due to symmetric positions and other advances the game was already solved in 1980.

1987 Connect4: Allis [2] and Allen [1] worked independently on the solving of Connect4. Both succeeded in the same month in 1987 with two distinct different approaches. Allen highly optimized the heuristics to reduce the runtime of the Alpha-Beta Search and Allis used a knowledge based approach (defining and proving rulesets) in combination with an Conspiracy and depth-first Search.

1994 go-moku: go-moku (Connect5) was also solved by Allis in 1994 [4]. He made use of a threat based approach which again radically reduced the search space and let to the solution of go-moku.

1996 Nine Men's Morris Gasser[16] combined the construction of an endgame database with an Alpha-Beta Search so solve Nine Men's Morris in 1996. Gasser stated, that "Nine Men's Morris is the first non-trivial game to be solved that does not seem to benefit from knowledge-based methods".

2007 Checkers: Schaeffer et al. [32, 33] began in 1989 to develop their checkers AI, called *chinook*. In 1992 they won the first man-machine-world-championships. Since then their program was above grand-master level. But they continued the development for the next years[31], utilized a cluster to build an endgame database[34] and finally prove the draw in 2007 [30]. All together they worked 18 years to solve checkers. Nevertheless Schaeffer stated that nowadays the solution would become possible within less than a few years.

3.2 Definition of solving

Without any formal description most people would probably say that a game is solved, when you know how to play to win in every match. However it is often out of sight, that the opponent may also play perfect and therefor a win is not always possible. Hence, informally speaking: a game is solved, when you know how to achieve your best possible result, even if it is a draw.

Over the years, the definition of solving became clearer, and the most accepted definition of solving is divided into three levels of solving. Victor Allis [4] counted the following terms:

- **Ultra-weakly-solved** The perfect-play result for the initial position(s) is known, but the strategy to achieve this result is not determined. One common example is *Hex*, which is proved as a first player win, but until now, no perfect strategy exists [43].
- **Weakly solved** A strategy for the initial position(s) is known to obtain at least the perfect-play result, for both players. Several games have been weakly solved, for example Connect Four [1, 2], Go Moku [4], Nine Men's Morris [16] and Checkers [30].
- **Strongly solved** A strategy to obtain at least the perfect-play result, for all legally reachable positions is known. Examples are Tic-Tac-Toe (solveable on a single sheet [27]), Awari [7] and chess endgames.

It is quite obvious that a *strongly* solved game is also *weakly* solved, since strategies for all positions are known also for the initial position(s). Moreover *strongly* and *weakly* solved games are as well *ultra-weakly* solved because a strategy is known the gain the perfect play result. The optimal play result is further denoted as *game theoretical value*.

3.3 The task

The remainder of this thesis deals with the solvability of Pentago with the best case goal to *weakly* solve the game under reasonable resource constraints. Reasonable resource constraints are that the program which solves Pentago should run on actual consumer hardware or a small cluster (below 50 instances). Furthermore the computation should finish in not only a predictable but also in a reliable amount of time, e.g below one year.
Therefore after evaluating the complexity of Pentago, several algorithms to *weakly* solve a game are discussed. To anticipate the results, we even were not able to *ultra-weakly* solve the game. But we give a detailed discussion of the occurred challenges and an insight why Pentago is not solvable on todays hardware. We also show, which technical means would be required to solve Pentago. We decided to implement one method and this practical part is “recycled” as an Artificial Intelligence for playing Pentago. The term ”recycled“ is used in the case, that our implementation does not fulfill its solving purpose.

4 Complexity of Pentago

In their survey paper on solved and solvable games Van den Herik et al.[43] mention two important measurements to estimate the solvability of a board game within a reasonable time: *state space complexity* and *game-tree size*. In the following sections both measurements are applied to the game of Pentago.

4.1 State space complexity

Definition 1. *The state space is the set of all, by legal moves reachable states of the game.*

A single state of Pentago can be described as a vector with 36 elements. As an example $s_1 = 0000\ldots0000$ represents an empty board while $s_2 = 0000\ldots0012$ can be interpreted as a board with a black and a white marble on $(5,6)$ and $(6,6)$. A formal description should usually contain the mapping between the vector position and board position, here omitted.
A naive approach to estimate the size of the state space n is to assume that there are three configurations per field (empty, occupied by white, occupied by black) and 36 fields per board, hence

$$n_{naive} = 3^{36} = 1.5 \cdot 10^{17}.$$

Using this simple combinatorial approach, states like $s_3 = 1111\ldots1111$ (only white marbles on the board) are also considered which obviously can not occur by legal movement. So n_{naive} is not the correct size of the *state space complexity* but offers a first upper bound.
Counting only states where the number of marbles on both sides is equal or differs by one will give a better approximation.

Definition 2. *For $n, k_1, k_2, \ldots, k_m \in \mathbb{N}$ with $n = k_1 + k_2 + \cdots + k_m$ the multinomial coefficient is defined as*

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1!\,k_2!\cdots k_m!}.$$

The multinomial coefficient is the generalization of the binomial coefficient. Given a set of n distinct objects, the multinomial coefficient expresses the number of possibilities to select $k_1, k_2, \ldots, k_m$ objects without regarding the selecting order in $k_1, k_2, \ldots, k_m$.
Giving an example, to calculate the possibilities of the German lottery game "6-aus-49", setting m to two, $k_1 = 6$, $k_2 = 49 - 6 = 43$, the multinomial coefficient results in $\frac{n!}{k_1!k_2!} = \frac{n!}{k_1!(n-k_1)!}$ which is equal to the definition of the binomial coefficient $\binom{n}{k_1} = \binom{49}{6}$.
Using the multinomial coefficient, we are able to appraise a harder upper bound on the possible game states. Viewing the board as a bag filled with markers for the three different configurations (empty, white and black), a state vector can be filled by picking these markers. The number of all combinatorial possible vectors can now be calculate via the multinomial coefficient.
Let k_b be the number of marbles placed on the board of the first player, k_w the same for the opponent and k_e the left empty spaces on the board. Pentago is typically played on a 6x6 board, so obviously $n = k_b + k_w + k_e = 36$. Beginning with an empty board $k_b = 0, k_w = 0, k_e = 36$, we alternately increase k_b or k_w by one and decrease k_e to cover all possible game situations. The results are shown in Table 4.1.

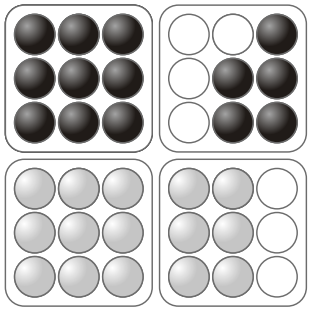


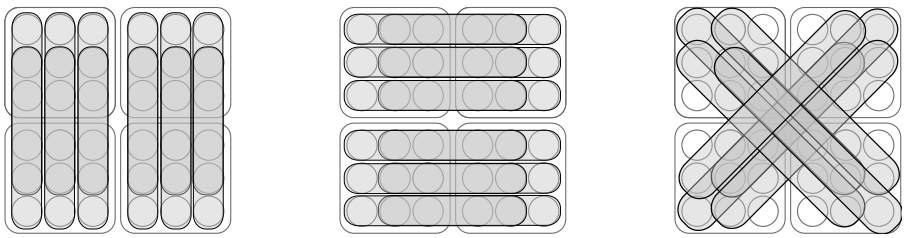
Figure 4.1: An impossible position, since all predecessors had to be a win for white.

The approximation of the overall *state space complexity* decreases in comparison to n_{naive} by the order of one magnitude to:

$$n_{multinomial} = \sum \binom{n}{k_b, k_w, k_e}$$
$$n_{multinomial} = 2.4 \cdot 10^{16}.$$

Unfortunately these results still overestimate the number of reachable positions. The naive usage of the multinomial coefficient does not regard further game mechanics, like the impossibility of two winning lines in all four quadrants, illustrated in Figure 4.1. It is impossible to reach this state, because all predecessors have to be a winning state for one of the players.

To get an exact result for the *state space complexity* of Pentago, a far more complex algorithm is needed to regard all unreachable positions, which will require a recursive visiting of all positions and summing them up. To overcome this problem, we show in the following paragraph, that the dimension of our estimation does hardly differ from the exact dimension.
Let n_{win} be the number of winning states within the *game state space*. The *multinomial coefficient* can once again be used to get an estimation of n_{win}. The ninth move is the first possibility for player one to win a Pentago game. Since then, it is possible to estimate an upper bound onto the n_{win} by fixing five stones of one color and calculating the multinomial coefficient for a board with $f = 36 - 5 = 31$ fields. A winning line can be on 12 vertical, 12 horizontal and 8 diagonal positions $l = 12 + 12 + 8 = 32$; explained in Figure 4.2.



(a) Five in a row vertical

(b) Five in a row horizontal

(c) Five in a row diagonal

Figure 4.2: 32 possibilities to achieve five in a row in Pentago.

So the result of the multinomial coefficient has to be multiplied by 32. Given move k with k_b black and k_w white marbles the number of winning states has to be lower than:

Table 4.1: Upper bound of state complexity for Pentago.

black	white	empty	#total positions p	$log_{10}(p)$	$log_2(p)$
1	0	35	36	2	5
1	1	34	1260	3	10
2	1	33	21420	4	14
2	2	32	353430	6	18
3	2	31	3769920	7	22
3	3	30	38955840	8	25
4	3	29	292168800	8	8
4	4	28	2118223800	9	31
5	4	27	11862053280	10	33
5	5	26	64055087712	11	36
6	5	25	277572046752	11	38
6	6	24	1156550194800	12	40
7	6	23	3965314953600	13	42
7	7	22	13028891990400	13	44
8	7	21	35829452973600	14	45
8	8	20	94052314055700	14	46
9	8	19	209005142346000	14	48
9	9	18	441233078286000	15	49
10	9	17	794219540914800	15	49
10	10	16	1350173219555160	15	50
11	10	15	1963888319352960	15	51
11	11	14	2678029526390400	15	51
12	11	13	3124367780788800	15	51
12	12	12	3384731762521200	16	52
13	12	11	3124367780788800	15	51
13	13	10	2643695814513600	15	51
14	13	9	1888354153224000	15	51
14	14	8	1213941955644000	15	50
15	14	7	647435709676800	15	49
15	15	6	302136664515840	14	48
16	15	5	113301249193440	14	47
16	16	4	35406640372950	14	45
17	16	3	8330974205400	13	43
17	17	2	1470171918600	12	40
18	17	1	163352435400	11	37
18	18	0	9075135300	10	33
Total			**24,072,650,378,629,800**	16	54

$$
\begin{aligned}
n_{win}(k_b, k_w) &= n_{winBlack}(k_b, k_w) + n_{winWhite}(k_b, k_w) \\
n_{winBlack}(k_b, k_w) &= l \cdot \binom{31}{k_b - 5,\ k_w,\ 36 - k_b - k_w} \\
n_{winWhite}(k_b, k_w) &= l \cdot \binom{31}{k_b,\ k_w - 5,\ 36 - k_b - k_w}.
\end{aligned}
$$

Table 4.2 shows n_{win} in comparison with $n_{multinoimal}$. The estimation of the number of winning states is really rough, but n_{win} and $n_{multinomial}$ differ by at least one order of magnitude, hence the approximated size of the *game state space*, $n_{multinomial}$, should contain an error around ten percent. Here we assume that all states misleadingly included in $n_{multinomial}$ should also be counted in n_{win}, except there exists a not winning but unreachable game state. We are not able to construct such an example.

Table 4.2: Comparison of winning states versus all states.

black	white	empty	#total positions	#winning positions
5	5	26	$6.4 \cdot 10^{10}$	$9.5 \cdot 10^{6}$
6	5	25	$2.7 \cdot 10^{11}$	$1.4 \cdot 10^{8}$
⋮	⋮	⋮	⋮	⋮
12	12	12	$3.3 \cdot 10^{15}$	$4.0 \cdot 10^{14}$
13	12	11	$3.1 \cdot 10^{15}$	$4.9 \cdot 10^{14}$
13	13	10	$2.6 \cdot 10^{15}$	$5.1 \cdot 10^{14}$
14	13	9	$1.8 \cdot 10^{15}$	$4.6 \cdot 10^{14}$
⋮	⋮	⋮	⋮	⋮
18	17	1	$1.6 \cdot 10^{11}$	$1.8 \cdot 10^{11}$
18	18	0	$9.0 \cdot 10^{9}$	$1.1 \cdot 10^{10}$
Total			$2.4 \cdot 10^{16}$	$3.3 \cdot 10^{15}$

Summing up, $n_{multinomial} = 2.4 \cdot 10^{16}$ provides a sufficient approximation of the *state space complexity* to draw further conclusions onto suitable solving algorithms.

Axial symmetry

The *state space space* can be reduced by using symmetry. As shown in Figure 4.3, there are four axis of symmetry. A lookup table for any solving approach may use the symmetry properties to map up to eight positions onto a single slot to save memory. However, mapping a position towards another requires CPU load, which has to be considered into a time-memory trade-off. Experimental results for utilizing symmetry are shown in Chapter 6. Quite similar observations regarding the symmetry of Pentago are given in [37].

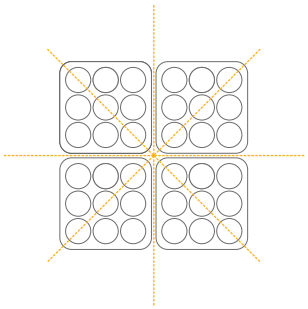


Figure 4.3: Axial symmetry of Pentago Four axis of symmetry exist. One horizontal, one vertical and two diagonal.

4.2 Game-tree size

Next to the state space the size of the game-tree is an important measurement to approximate the solvability of a board game. The game-tree and the size of the game-tree are defined as:

Definition 3. *A game-tree is a directed graph composed of game states as nodes and moves as edges. A complete game-tree is a game-tree beginning with the initial state and containing all possible moves.*

Definition 4. *The game-tree size is the number of leafs of the complete game-tree, equal to the number of possible games that can be played.*

The most meaning full parameters to estimate the *game-tree size* are the number of moves until a game finishes, called game depth, and the number of possible moves per state. To get the first upper bound onto the *game-tree size*, the maximum game depth can be regarded as $d = 36$. After the $36th$ marble is set, no more moves are possible, the game is a draw. The number of possible moves is bounded by

$$M(d) = freePieces \cdot quadrants \cdot directions = (36 - d) \cdot 4 \cdot 2.$$

Taking these numbers into account, an upper bound for the *game-tree size* can be calculated by

$$n = \prod_{i=1}^{36} M(i) = 5 \cdot 10^{63}.$$

A common approach to estimate the *game-tree size* for game solving is the averaging of game depth, especially if the game is a win for one player, than the game will end before the 36th move. Moreover many moves from one state fabricate the same successive state. Often database of players on grandmaster level, combined with an evaluation of existing computer AIs are used to calculate reliable average values[30, 43]. Until now to our knowledge no database of played Pentago games exists. Our own experiments with *PentagoAI* showed the following results:

$$depth_{avg} = 21$$
$$n = \prod_{i=1}^{21} M(i) = \frac{36!}{15!} \cdot 8^{21} = 2 \cdot 10^{48}.$$

This value is more reliable the the first mentioned $n = 5 \cdot 10^{63}$, therefor further conclusions will be based on the experimental observed *game-tree size* of $n = 2 \cdot 10^{48}$.

4.3 Conclusions on the complexity of Pentago

Looking at Table 4.1, the size of the state space increases from move to move until the endgame is reached. Beginning with an empty board, pieces are added during the play. This is opposite to convergent games, where pieces are gradually removed during the game. Therefor Pentago is a divergent game. The size of the state space increases in divergent games.
Typically convergent games are the target for solving on a state space approach. These approaches are also mentioned as *brute-force* methods [43] because they often do not make use of any selective search, rather than searching the whole state space. If the size of the state space is suitable, the *brute force* methods will succeed. Even so Pentago is a divergent game, Section 5.4 in Chapter 5 discusses the suitability of one *brute force* approach on the state space.

Table 4.3: Complexity of different board games.

Game	State space complexity	Game-tree complexity	Solved	Reference
Qubic	10^{30}	10^{34}	Yes - 1980	[3, 19, 26]
Connect4	10^{14}	10^{21}	Yes - 1989	[1, 2, 19]
Go-moku	10^{105}	10^{70}	Yes - 1994	[4, 19]
Nine Men's Morris	10^{10}	10^{50}	Yes - 1996	[16, 19]
Awari	10^{12}	10^{32}	Yes - 2003	[19]
Checkers	10^{21}	10^{40}	Yes - 2007	[30, 35]
Pentago	10^{16}	10^{48}	No	-
Chess	10^{46}	10^{123}	No	[43]
Shogi	10^{71}	10^{226}	No	[43]
Go	10^{172}	10^{360}	No	[43]

To give the pure numbers on complexity of Pentago some comparison, Table 4.3 shows the complexity of other board games. As described by Heule and Rothkrantz [19] and Herik et al. [43] there is some correlation between the state-space and game-tree complexity and the solvability of a game. *Qubic* and *go-moku* are special cases, which have been solved by knowledge-based methods which cannot be easily applied to other games. Both games have a rather small decision complexity. Heule and Rothkrantz described the decision complexity, as the number of decisions which have to be stored for a full solution. This might be all the nodes in a solution tree but also a proved pattern database or a symmetric position may decrease the decision complexity. The decision complexity of Pentago is not easily measurable but our observations did not show any sign of a low one. Herik et al. introduced a classification of solvability according several methods and complexity. Later Heule and Rothkrantz refined these classification which is illustrated in Figure 4.4. They stated that games with a lower state space complexity than 10^{10} or game-tree complexity of 10^{20} are solvable by any brute-force methods. All others games have to be checked in detail for applicable algorithms. Since Pentago cross both lines further investigations in solving methods are required. Those are given in the next chapter.

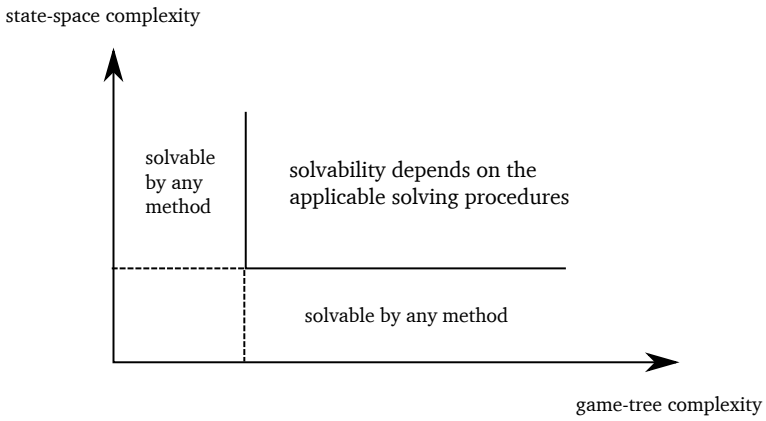


Figure 4.4: An (alternative) view on the game space. Figure by Heule and Rothkrantz [19].

5 Solving methods and algorithms

Within the last 50 years several methods have been developed to solve games. In this chapter we discuss the main common solving procedures for zero-sum two player games with perfect information [19, 43]. It is obvious, that given an infinite amount of time, every game with a finite number of moves is solvable by walking through the full *game-tree*. Since we do not have infinite time for solving games, we have to find procedures, which reduce the problem and the solution size to acceptable computational costs.
The presented procedures are:

- *Alpha-Beta Search and variants*
- *Proof-Number Search and variants*
- *Threat-space Search*
- *Retrograde Analysis*

All procedures in this chapter are introduced in general, but with a special focus on the computational costs for reducing the size of the solution. Moreover we point out their applicability onto Pentago. The procedures are categorized, according their search direction, either forwards (Sections 5.1,5.2,5.3) or backwards (Section 5.4)). The combination of both approaches is discussed in Section 5.5. Section 5.6 gives a conclusion of the presented algorithms.

5.1 Alpha-Beta Search

The Alpha-Beta Search, also known as Alpha-Beta pruning algorithm is probably the most common choice for implementing game AIs. The Alpha-Beta Search is an adversarial depth-first Search algorithm and in principal an extension of the minimax algorithm [20]. It is based on a brute-force search through the game-tree by pruning unnecessary sub-trees.

To explain the Alpha-Beta Search algorithm, we start with a repetition of the minimax algorithm. Given a position, minimax evaluates this position by expanding a game-tree until depth d. The values of the leafs are set according a game-specific *evaluation function*. In general a winning position gets a high rating, a loosing positions a low rating and all positions inbetween are heuristically rated. For example a material advantage of a queen in *chess* gives typically a better rating than material equal positions. Given this tree, all internal values are set either to the maximum or to the minimum values of their children. This depends, whether it is the turn of the first player (MAX) or his opponent (MIN). The procedure is recursively repeated until a value is propagated to the root. Two important remarks are, that this algorithm can be run in a left-most depth-first manner, so that only a single branch has to be maintained in memory. And second, the result of the minimax algorithm is perfect in the case that the evaluation function gives the correct game theoretical value. Hence, minimax can be used to solve games.

Alpha-Beta Search tries to avoid needless visits of nodes which do not have an influence on the whole result. For example: a position is given with n possibilities to move for the MAX player: $m_1, \ldots, m_n$. Now the MAX player evaluates the consequences of move m_1 and receives as a result, that he will win the game, if he plays move m_1. Since he already knows which move leads to the highest result possible he does not need to investigate the moves $m_2, \ldots, m_n$. So without any consequences for the game

theoretical result, Alpha-Beta Search finds such situations and prunes unnecessary sub-trees. To afford the goal effectively the Alpha-Beta Search shrinks the branching factor by setting a lower (alpha) and an upper (beta) border. Every search in each sub-tree is bounded by the window between the alpha and the beta value. The alpha value presents the actual best possibility for the MAX player and the beta values presents the same for the MIN player. Initially alpha is negative infinity and beta is positive infinity. The window narrows typically through the recursive process. If a branch of a MIN node is evaluated below the alpha value, than all other branches of the MIN node are pruned. This is because the actual best move for the MAX player in an upper part of the three is better than the possible outcome of this MIN node. This behavior is called Alpha-Cut-Off. There is also a Beta-Cut-Off, when a move in a MAX node returns a higher evaluation than the beta border, all the remaining moves in the MAX node can safely be pruned. Figure 5.1 explains the Alpha-Beta Algorithm among a detailed example. Another good description of the Alpha-Beta Search algorithm can be found in [28].

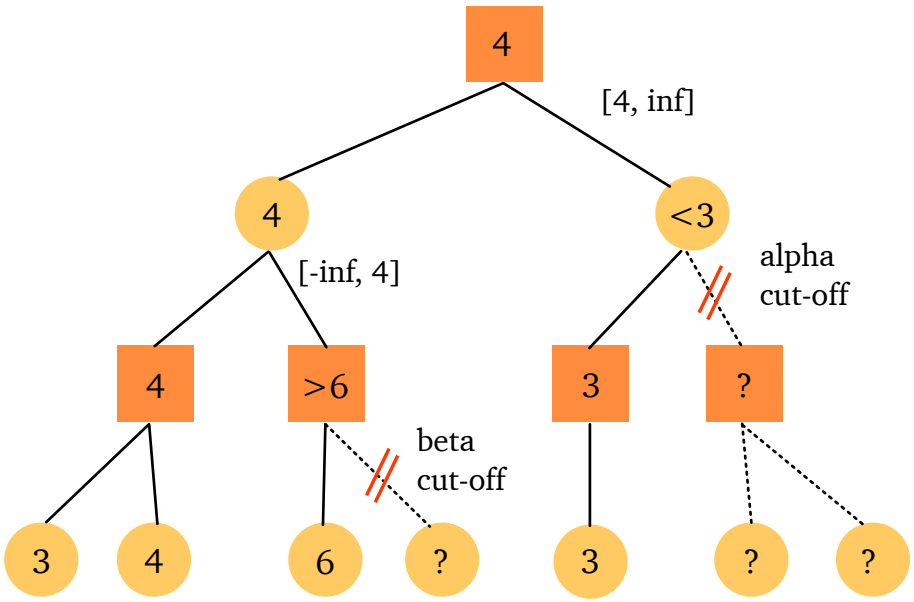


Figure 5.1: Alpha-Beta Search algorithm: This is an example for an evaluated tree with the Alpha-Beta Search algorithm. MAX nodes are denoted as squares and MIN nodes as circles. The leafs values are given by an evaluation function and propagated trough the inner nodes to the root. Since Alpha-Beta Search is an depth-first algorithm the search begins at the deepest left-most leaf and its sibling. The max value (4) of both leafs is given to the MIN node above. The second branch of the MIN node is now evaluated with the upper border (beta) of 4. The first leaf reveals a value of $6 > 4$ which leads to a beta-cutoff because now this branch of the MIN node reaches at least the value of 6 which is definitively worse than the first branch. The same occurs in the examination of the second branch of the root (MAX).

5.1.1 Variants and improvements

Alpha-Beta Search was a research subject for many years, so a large number of improvements and variations have been developed. We will discuss the most common ones which do not violate the correctness criterion. Many variants allow a faster search based on heuristics but they might oversee some positions which is not acceptable for a solving algorithm.

Move ordering: As stated before, the Alpha-Beta Search algorithm returns exactly the same result

as the minimax algorithm. But move ordering has a huge influence on the runtime of the Alpha-Beta Search whereas it does not influence the runtime of the minimax algorithm. Since the window between the alpha and beta value only shrinks if better moves are found, the Alpha-Beta Search performs the same as minimax if the worst moves are the first branches. Now it becomes impossible to prune any node. In consequence it is quite useful to order all moves by using a heuristic before examining them to reduce the complexity of the Alpha-Beta algorithm although move ordering itself might be costly.

Transposition table: As in several other board games many equal positions are reachable through different paths in the game-tree. So it might be useful to save the result of any evaluation in a so called transposition table. Everytime the Alpha-Beta Search algorithm encounters this position again, it just takes a look in the transposition table to rate this position. Typically the transposition table is organized as a hash-table to offer fast lookups. Since not all positions fit into a transposition table within the main memory, strategies are required to share the available space [9, 41]. It is important to note, that the usage of transposition tables transforms the search-tree into a search-graph.

Iterative deepening: Instead of searching until depth d, the search runs d times with the search depths $1 \dots d$, beginning with depth 1. Although this iterative process requires more computational costs in theory, it has been noticed that iterative deepening is often faster than searching for the given depth. This happens due to dynamic move ordering techniques which use the results of the former search [36]. Another feature is, that the search can be aborted by resource restrictions, but still returns a useful output of the previous iteration.

Variable search depth: The outcome of all moves is not equal. Some moves lead to new complex situations and require further exploration and others are more quite moves. Hence it might be useful to reduce the search depth for weak moves and increase it for strong moves instead of a fixed-depth search. This might not be exercisable for a holistic solving approach, since the whole tree has to be evaluated.

Minimal and Aspiration window: It has been shown [36, 38], that it might be useful to choose a smaller window size than the Alpha-Beta Search suggests. On the one hand this leads to a raised number of cut-offs on the other hand, it has to be checked, whether the search with the narrowed window was correct. If the assumption was wrong, the search has to be restarted with a larger window size.

This list of advances in the Alpha-Beta Search is by far not complete. Further ideas are noted in [5, 36, 38].

5.1.2 Computational costs

To solve a game with minimax search algorithm, the whole game-tree has to be evaluated. This results in a complexity of $\Theta(moves^{depth})$, where $moves$ and $depth$ are observed average values, also described in the prior Chapter 4. But Alpha-Beta Search is able to reduce the exponent by half if perfect move ordering is given: $\Theta(moves^{depth/2})$ [20, 28]. Even with random move ordering the complexity shrinks to $\Theta(moves^{depth \cdot 3/4})$. Transposition tables further decrease the complexity, but it is harder to estimate their effort.

5.1.3 Applicability onto Pentago

In the last chapter we showed, that the size of the game-tree is estimated with $2 \cdot 10^{48}$. Thus, given perfect move ordering the computational costs for the Alpha-Beta Search algorithm are around

$\Theta(moves_{avg} \cdot depth_{avg}/2 == 1 \cdot 10^{24}$. But on the negative side, there is no easy way to order the moves. Pentago allows a rotation in every move, so the board may totally change from move to move. This possibility has to be taken into account for designing a reliable heuristic.
Furthermore we do not have access to human expert knowledge in Pentago, which could be used to set an good initial line to increase the pruning factor since good branches occur at first [35].

Time approximation: Note: The following approximation is really rough and disregards transposition tables. We assume costs of 500 instructions to examine an internal node and to generate all successor moves in nearly perfect order. We further estimate the costs of the evaluation function with 100 instructions. The evaluation of the leafs is rather cheap, since we can efficiently check for five-in-a-row, using bitboard techniques. In comparison, the examination of an internal node requires an expensive move and successor generation. Our time estimation is based on the performance of an actual Intel Core-i7 990X @3,46 GHz with 146,000 MIPS [40]. All together this leads to the following time approximation:

$$
\begin{aligned}
internalNodes &= 1 \cdot 10^{22} \\
leafs &= 1 \cdot 10^{24} \\
instructions &= internalNodes \cdot 500 + leafs \cdot 100 \\
ips_{i7} &= 146,000\,\text{MIPS} \\
time &= \frac{instructions_{total}}{ips_{i7} \cdot 10^6} = 3,5 \cdot 10^{15}\,\text{seconds} = 10^8\,\text{years}
\end{aligned}
$$

This totally exceeds by far our time restrictions. On the positive side, it might be possible, that the game is a forced win for the first player before the 36th move. This could decrease the average game depth, so a valid solution becomes possible. Moreover the board of Pentago is rather small, so utilizing a transposition table might have a large influence on the complexity. Without any of these assumptions Pentago is actually not solvable with the Alpha-Beta Search algorithm. However all these speculations require further evaluation with a testing Alpha-Beta implementation, then a clear statement may become possible.

5.2 Proof-Number Search

Proof-Number Search (PN) is a best-first search algorithm for the game-tree [42], introduced by Allis [4] to solve *Qubic*. PN Search is based on the work on Conspiracy Search by Schaeffer and McAllester [23, 29] and it is especially suited for solving endgames. The key idea behind PN Search is to prove the game-theoretical value by expanding nodes which heuristically promise the fastest proof.

A PN-tree can have three values: *true*, *false* and *unknown*. PN Search focuses onto a proof or disproof of a binary property, for example a forced win for the first player denoted as *true*. Hence, a disproved tree does not necessarily show a forced win for the opponent, the game theoretical value might also be a draw. If no proof or disproof is possible, the value of the tree is *unknown*.
PN works as follows, the search-tree maintains two numbers for every node, the proof number *pn* and disproof number *dn*. Both numbers express the estimated or calculated number of leafs which are further required to prove or disprove the property for the given node. Thus, a winning leaf for player one, will get the proof number $pn = 0$ since no more expanding of nodes is required to prove the win. The disproof number for a winning leaf is set to $dn = \infty$, since it is impossible to disproof this node. Unknown

leaf nodes are given the numbers $pn = dn = 1$, but it is also possible to assign them values based on heuristic estimations, for example evaluation functions. The pn / dn values at the leafs are propagated according AND / OR rules. A MAX move for the first player is represented by an OR for the pn number, thus the minimum of the children $pn_{father} = min(pn_1, \ldots, pn_n)$ and the dn number is aggregated by an AND which is the sum $dn_{father} = pn_1 + \cdots + pn_n$. A MIN move works vice versa to the MAX move. Now given this AND / OR tree, PN-Search crawls the tree by expanding the *most proofing node*. Beginning at the root (OR node), the branch with the lowest pn value is chosen, followed in the next level (AND node) with the choice of the lowest dn value until the *most proofing node* is recursively identified. Hence, PN Search is a best first search. Figure 5.2 presents an example of the PN Search.

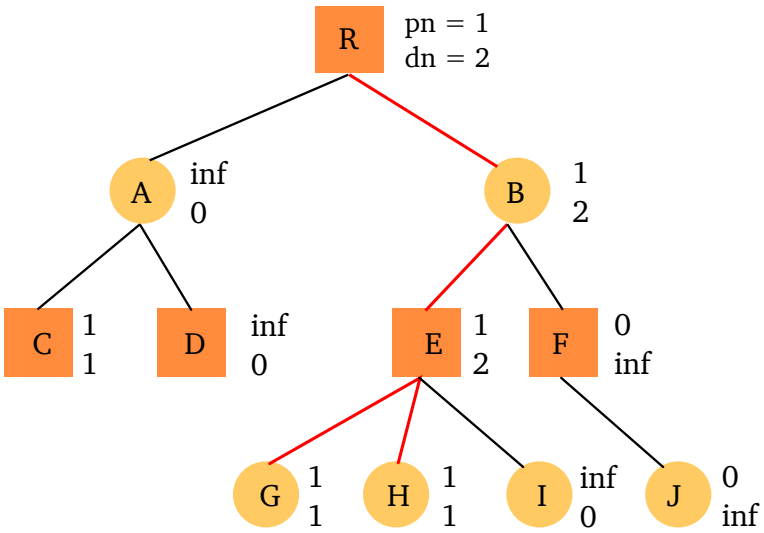


Figure 5.2: Proof-Number Search This is an example of a Proof-Number Search tree. A square denotes a MAX / AND node, whereas a circle denotes a MIN / OR node. The proof and disproof numbers are written next to each node. The leaf **J** is proved as true, therefor its pn value is 0 and its dn value is ∞. Opposite to that node, **D** and **I** have been disproved. The way to the two most proving nodes is illustrated through the red bold line. In each AND node the pn value is the minimum of its children pn values and the dn values is the sum of the children dn values. Vice versa the same holds for the OR node. Here the pn value is the sum of the pn values of its children and the dn value is the minimum of the children dn values.

5.2.1 Variants and improvements

What is the disadvantage of best-first search algorithms? It is unlikely to fulfill the memory requirements, because the whole game-tree has to be stored in the main-memory. At least this assumptions holds for games which are non-trivial to solve. Because of this problem a few ideas and variants of the PN Search have been developed. We do not discuss them in detail, but their basic ideas and consequence on the computational requirements are described:

Removing sub-trees: Allis suggested that all (dis)proved sub-trees may be deleted, since only their result is relevant and no further investigation within these sub-trees is necessary. This may dramatically shrink the tree size and in consequence the memory requirements. But an important remark is, that the *weakly/strongly solved* property is violated because the strategy to play within these deleted sub-trees becomes unknown. However during runtime playing, sub-trees which are rather small in comparison to

the whole game-tree could be reconstructed with adequate costs.

PN^2: Allis also suggested the PN^2 Search algorithm whose idea is based on a two level PN Search. The first level PN Search (PN_1) initiates the second level PN Search (PN_2) to evaluate every most-proving node. The second level PN Search is limited by the number of positions which can be stored. This limitation is set either by the bounds of the memory capabilities or by setting an upper bound through the PN_1 Search. Hence, the PN_2 Search can prove, disprove or abort its sub-tree. Proved and Disproved sub-trees are removed, only their results are preserved. This two level approach is called *delayed evaluation*, since all children revealed in the PN_1 Search are not directly evaluated, only the most-proving node is passed to the PN_2 Search.

PDS Nagai proposed the *Proof-number and Disproof-number Search* (PDS) as a multiple-iterative deepening PN algorithm. PDS iterates at all nodes, not only beginning at the root like in typical iterative deepening algorithms. Every node has two thresholds, one for the proof and one for the disproof number. Now each sub-tree is searched until the proof or disproof number is below the according threshold. Moreover PDS makes extensive use of transposition tables to further decrease the runtime. Herik et al. [42] showed, that PDS is able to solve harder problems than the PN Search, although it is not faster.

A few other minor variants exist, like the *depth-first proof-number search* (df-pn). But for our considerations these algorithms do not make larger difference than the already presented variants.

5.2.2 Applicability onto Pentago

Obviously PN Search is memory bound. In chapter 4 we estimated the size of the game-tree with $1*10^{49}$ and a naive implementation of a best-first search algorithm would probably exceed all memory resources. But the depth-first variants of PN Search might handle this large game-tree size, especially when the solution tree is much smaller than the full game-tree.
PN Search showed the most advantage in comparison to Alpha-Beta Search in games with unbalanced trees [42]. On perfectly balanced trees PN Search did not perform any better than an optimized Alpha-Beta Search algorithm, sometimes even worse. Since we do not have any expert knowledge on Pentago or a tournament game database, we can only guess about the uniformity of the game-tree. Our own observations showed that often positions occur which force the opponent to react to avoid losing the game. This fact might lead to an uneven tree. But to prove the existence of an unbalanced tree, a first implementation is required. (The concept of a forced move is also discussed in the next section on *threat* based search algorithms.)
Moreover the most successful PN Search implementation relied on a good heuristic to rate new leafs. But finding a good heuristic for Pentago is really hard. This problem is a similar problem to that in the evaluation function in the previous section about Alpha-Beta Search. The only useful approach to evaluate a node requires either a large pattern database or an expanding of the nodes children. Both suggestions do not seem very promising according there computational costs, so in consequence a first implementation of the PN Search has to be done without any heuristically rated leafs.

Summing up it seems at least worth trying to evaluate a PN Search by using a variant with reasonable memory requirements. A first evaluation of the PN Search could begin with a proof of advanced positions, where the Search depth is bounded by the number of empty places. With these results a more exact prediction about required memory, time and tree uniformity should be possible. If the tree shows up to be non-uniform and the memory within a small cluster is large enough to handle the PN Search

Table 5.1: Complexity of qubic and go-moku and their solution sizes.

Game	solved in	search-space complexity	game-tree complexity	solution size
qubic	1980	10^{30}	10^{34}	2929
go-moku	1994	10^{105}	10^{70}	138.790

than the algorithm can be enlarged to earlier positions to finally solve Pentago.

Time approximation: Without any knowledge of the game-tree we can only speculate about the computational time requirements to solve Pentago. Given enough memory resources a depth-first or iterative PN implementation will probably be faster than the Alpha-Beta Search. But only a speed-up of more than seven orders of magnitudes in comparison to the Alpha-Beta Search brings the PN Search into reasonable regions of computational costs as of today.

5.3 Threat-space Search

Threat based search algorithms reduce the problem size by focusing on *game threats*. A *game threat* forces the opponent to react and to concentrate his possibilities on the threat. A classic example for a threat is the check in chess. Here a player is even forced by the rules to counter the check. Thus the opponent is limited in the number of moves whose he is allowed to do. Instead of maintaining fully expanded sub-trees, threat-based search algorithms focus on saving and traversing threat-sequences which are often much more smaller.

According to Heule and Rothkrantz [19], at least three different variants of thread based searches exists. They distinguish between *Threat-sequence Search*, *Threat-space Search* and *Lambda Search*. In the following examination of the different algorithms, we disregard *Lambda Search* since it is only for games where passing moves is allowed. Despite the fact that both remaining algorithms have several differences we discuss them together, because they have similarities according their applicability onto Pentago.

Threat-sequence Search was introduced to solve qubic [26], whereas **Threat-space Search** was pointed out to solve go-moku [4]. Both games have a large search-space complexity and also a large game-tree complexity in comparison to other solved games so far. Anyhow qubic has already been solved in 1980 and go-moku in 1994. This is because the two algorithms excellently play out their possibility to reduce the solution size. See Table 5.1 for details.

Both algorithms search through the threat-space to find *winning-threat-sequences*. However if no threat occurs, they have to fall back on other search algorithms, like the prior mentioned Alpha-Beta or Proof-Number Search. Threat-sequence Search crawls through all combinations of threats until no new threat occurs. Hence, if any winning-threat-sequence is possible, then Threat-sequence Search will find it, even so this procedure might be cost intensive. In contrast, the Threat-space Search focuses rather on threats which will produce new threats. This approach reduces the number of branches, but also increases the chance to oversee winning sequences. Furthermore Allis executed all possible defending moves together to speed up the search, despite the fact that he might overlook even more winning sequences. This improvement might be hard to understand while first reading and requires a further insight in the game of go-moku. Go-moku played between advanced players typically results in a play where one player

plays a threat sequence until no further threat is possible and the opponent starts his own threat sequence. The non winning-threats, for example 3-in-a-row, can often be countered by multiple moves. The Threat-space Search executes all defensive placing of marbles together within a single move, so an uneven number of stones might be on the board. Because of this, only a single branch for the opponent has to be considered. As described before. this reduces the search space but increases the chance to oversee an early win. Heule and Rothkrantz[19] described Allis approach as a more human like threat search. Allis [4] proved the correctness of his advances with a formal model in his thesis, here omitted since it goes beyond the scope of this work.

5.3.1 Applicability onto Pentago

Since go-moku and Pentago have several similarities, threat based algorithms seem to be very promising. It might be possible to implement threat based search algorithm to solve Pentago, however multiple disadvantages have to be studied. In principle we observed two reasons which argue against utilizing Threat-space and Threat-sequence for Pentago:

Finding threats in Pentago is very costly: Even if the board of Pentago is relatively small in comparison to go-moku, a large number of threats may occur and it is hard to describe the threats with simple patterns. To detect a threat, every possible move has to be evaluated, since each quadrant may be rotated within the next move. This is similar to expanding the search tree for the next level, which probably will be implemented faster within highly optimized tree searches like Alpha-Beta Search.

Large number of counter moves are possible: The second argument has an even larger impact on the applicability. Most threats can be countered by many moves. Figure 5.3 explains this fact among an example position. In many cases a threat may be countered by rotation a single quadrant. Hence, placing the marble is not concerned by the threat and the direction of the rotation is also irrelevant. Thus, a large number of moves is still playable and threat based search reduces the problem size by only a small factor. Furthermore it is not possible to play all counter moves together, as Allis used it for the solution of go-moku. This is because every move in Pentago also consists of a rotation and more than one rotation cannot be handled.

In our opinion, the trade-off that occurs while using Threat-space or Threat-sequence Search to reduce the problem size has to be considered carefully. A simple but easy and fast to implement game-tree search algorithm might outperform a more costly threat search algorithm. A strong conclusion requires an implementation and benchmarking of a threat based searches in comparison to other mentioned algorithms. Therefor we cannot give any reliable computational time approximation.

5.4 Retrograde Analysis

Retrograde Analysis is an undirected search through the *state space*, beginning at terminal positions towards the initial position.
Instead searching forwards through the game-tree, a Retrograde Analysis begins with all terminal positions and stores their game result (won, draw or lost) into a database. This is the base for the first iteration of the Retrograde Analysis. Assuming the first iteration of the Retrograde Analysis takes place at the deepest depth $d = n$ in the game-tree. In each iteration, the database for the previous depth $d - 1$ is constructed. If depth $d - 1$ is MAXs turn, every predecessor of a winning position at depth d is also

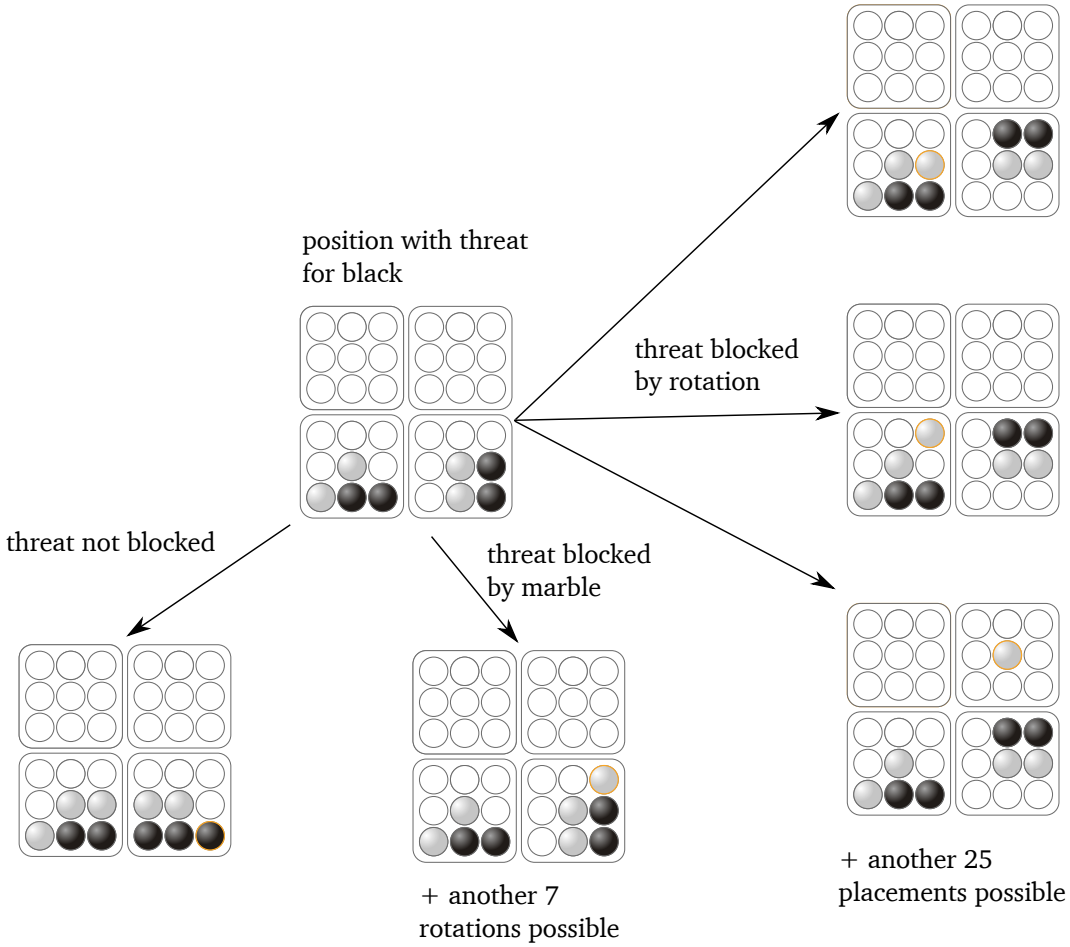


Figure 5.3: A position with a threat for black has to be countered by white.

marked as a win. Every unmarked predecessor of a draw positions at depth d is marked as draw and all the remaining unmarked positions in $d-1$ are marked as loss. This is vice versa for a MIN turn.
Two variants of the Retrograde Analysis are common, they differ in the way of constructing the database for depth $d-1$. At first, it is possible to iterate over the positions at depth d, as described before, to generate all prior positions and label them according the generating position. The second way is to iterate over the positions at depth $d-1$ and look up the successors of every position to gain the game theoretic value. For both variants the iterative process finishes at depth 0. Algorithm 1 illustrates this process.

Now the database contains perfect information for all possible game states, furthermore perfect play is possible. For each given position, the successor with the best game theoretic value in the database is chosen. This strategy does not guarantee a shortest move sequence, however a valid move sequence for the given value is known. A second search from the initial position through the constructed database is necessary to find the shortest move sequence for any position. Since Retrograde Analysis crawls through the whole *state space*, Retrograde Analysis strongly solves a game.

5.4.1 Computational costs

The whole *state space* has to be visited during a full Retrograde Analysis. This results in a complexity of at least $\Theta(stateSpace)$. For both mentioned variants of the Retrograde Analysis, the complexity is

Algorithm 1: A MAX step from depth $k+1$ to depth k in the Retrograde Analysis for games with deterministic game depth.
Input: Array with the game theoretical values of all positions at depth $k+1$
Output: Array with the game theoretical values of all positions at depth k

Input: $positions_{k+1}$
foreach p *in* $positions_{k+1}$ **do**
  $value_p \leftarrow$ value(p)
  $predeccessors \leftarrow$ predecessors(p)
  foreach $pred$ *in* $predeccessors$ **do**
    $hash \leftarrow$ hash($pred$,k)
    if $value(positions_k[hash])$ = UNKNOWN **then**
      $positions_k[hash] \leftarrow value_p$
    else
      $positions_k[hash] \leftarrow \max(value_p, \text{value}(positions_k[hash]))$
    end
  end
end
Output: $positions_k$

typically above $\Theta(stateSpace)$, because every lookup, either successor of predecessor, requires multiple access onto the lower level of the database to evaluate the MAX or MIN function. Consequently the average number of moves required to gain a MAX or MIN result has to be considered for the calculation of the complexity, $\Theta(moves_{avg} * stateSpace)$.

5.4.2 Improvements and practical issues

Parallel Retrograde Analysis: To decrease the runtime of the Retrograde Analysis an efficient parallel implementation is necessary. Applying parallelism does not decrease the computational costs, but the computational capacity can be increased. Larger implementations of a parallel database construction with Retrograde Analysis is given in [7, 17, 34].
To exploit parallelism it is important to decompose the database in several smaller sub-databases with as less dependency between the different sub-databases as possible. Independent sub-databases have two positive effects. On the one hand, they can be divided onto several processors or computers within a cluster and so computed independently, on the other hand smaller portions probably fit into a higher level of the cache hierarchy which could dramatically speed up the database accesses.

Compression: A second important improvement for the Retrograde Analysis is the usage of compression. As described, the constructed database requires at least one entry per state, which results in a memory consumption of $\Theta(stateSpace)$ entries. Therefor it is important to compress as many information as possible.
A naive database approach would save every state next to its game theoretic value. The state information in Pentago, requires at least a 6x6 board with 3 possible configurations per field. Even with Bitboard techniques (introduced in Chapter 6), a single state requires more than 4 Bytes of space. Because of this, typical Retrograde Analysis databases use *minimal perfect hash functions* [11] to map every state onto a concrete position in the database. *Minimal perfect hash functions* give a bijective mapping for every state $s \in S$ onto values between $0 \ldots |S|-1$. Hence the state information do not need to be stored inside the

database, only the game theoretic values of the states have to be saved.
Thinking in typical computational units (Char, Integer...), a game theoretic value requires at least a single Byte within the database. But the required information in the database just takes three values (won, loss, draw) or within the creation time a fourth value (unknown). Thus, 2 Bits are enough to save one state. Second, a good arrangement of the database offers the possibility to use a Huffman encoding (for example $010101\ldots010101$ is stored as $(N)x01$. A good arrangement means, that positions with similar game theoretic values should be located together, if any prediction is possible. To give an example, a decomposition in chess, based on the material could be useful. A player with an advantage of a queen will have a more likely chance to win than his opponent. Thus, this subset will have an increased number of win values, which leads to a better compression factor.

Both mentioned suggestions, have drawback on the computational costs, but Schaeffer et al.[21, 34] were able to run several machines in parallel and saved more than 20 positions in a single bit, which led to the solution of *checkers*.

5.4.3 Applicability onto Pentago

As seen before, the computational costs and memory consumption of the Retrograde Analysis relies on the *state space*. In Chapter 2 we estimated the *state space* of Pentago with $n = 2.4 * 10^{16}$. Unfortunately given actual consumer hardware, the Retrograde Analysis cannot easily be applied onto Pentago. This is because of multiple reasons:

Storage space: Even with an optimistic approximation, the amount of required storage space is close to the hardware borders. In Chapter 2 we showed, that 4 axes of symmetry exist, which results in a reduction factor of 8. Furthermore assuming a compression, comparable to *checkers*, 20 positions per bit, leads us to the following estimation:

$$\begin{aligned} \text{size}_{\text{db}} &= 2.4 \cdot 10^{16}/(8 * 20)\,\text{Bits} \\ &= 1.5 \cdot 10^{14}\,\text{Bits} \\ &= 1.875 \cdot 10^{13}\,\text{Bytes} \\ &= 17.5\,\text{TB} \end{aligned}$$

17,5 Terabyte of storage is nowadays feasible within a larger RAID or computer cluster, but this number still relies on an optimistic approximation. At first, a *minimal perfect hash function* which regards symmetry issues might be really hard to construct. Examples for *minimal perfect hash function* in similar board games without regarding the symmetry are shown in [13, 21]. But adding symmetry issues requires a unique ordering of all symmetry states and a mapping on continuous values, the latter problem is by far the complex one. Furthermore we disregarded any space for checksums, needed data-structures and file catalogs in our estimation. Summing up, the requirements on storage space might be accomplishable. However a good decomposition in independent parts is necessary, because the whole database will not fit onto a single Hard Disk (HD) or even Solid-State-Disk (SSD).

Database decomposition: As shown in Section 5.4.2 decomposition is essential for parallelism. A first and easy decomposition of the database can be based onto the number of marbles on the board. The sub-database with 4 marbles requires the sub-database with 5 marbles for construction purposes, but does not directly rely on the sub-databases with more than 5 stones. So this decomposition offers

a possibility to split the data onto several HDs, but it is still impossible to handle these different sub-databases in parallel. Moreover, according Table 4.1 in Chapter 4 the database with 24 marbles requires more than 10 percent of the whole storage space, which might fit onto a single HD but actual Random-Access-Memory (RAM) in this dimension, which would be much faster, is not payable. Hence further decomposition is needed to exploit parallelism and to fit the sub-database into the main memory.

Until this point, the application of the Retrograde Analysis on Pentago seems to be quite forward, but

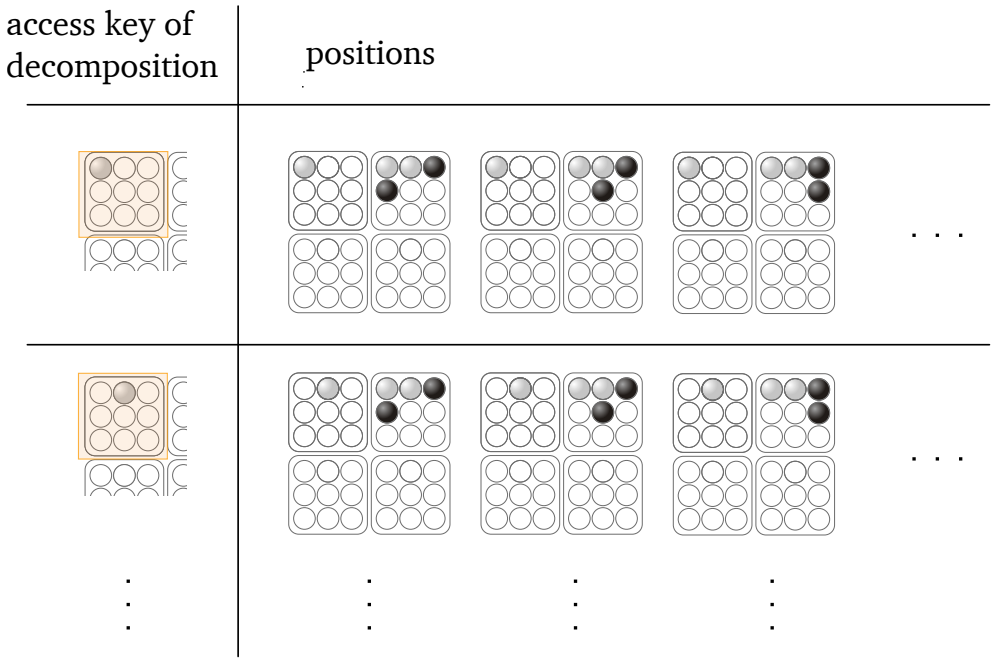


Figure 5.4: An exemplary decomposition of a database for Pentago. One quadrant is defined as the key quadrant and all positions according to this quadrant are handled together.

further decomposition is really hard, as a result of the rotating quadrants. For example subdividing the database according to a single quadrant might be possible. This single quadrant presents the index key for the decomposition. Figure 5.4 illustrates this process. Though any lookup for a single state, either forwards or backwards, within the Retrograde Analysis will require an access up to 8 different sub databases. From this follows a high interconnection between the different sub-databases. Other decompositions might be possible, but out of our knowledge we could not find any decomposition without this dependency issue.

Dependency and memory accesses: As described before with this high interconnection between the different sub-databases parallelism cannot be employed in a naive way, because the lookups into different sub-databases would create a large communication overhead. This overhead may either occur between different threads or between computers within a cluster, but the overhead may also result into many locks on shared memory machines. Thinking of $2.4 * 10^{16}$ states with an average of 180 possible moves per state will result in a huge amount of time waiting for synchronization, which is incomparable larger than straight forward processing states inside the CPU. Besides the synchronization time, the access times on the memory increase with every dependency. This is because of the impossibility to use cache localities. Register and L1-L3 caches are rather small and therefor many dependencies have to be solved through slow RAM or even HD accesses. Figure 5.5 explains the latencies and capacity differences between the diverse types of memory in the cache hierarchy. Perhaps a heuristic approach is able

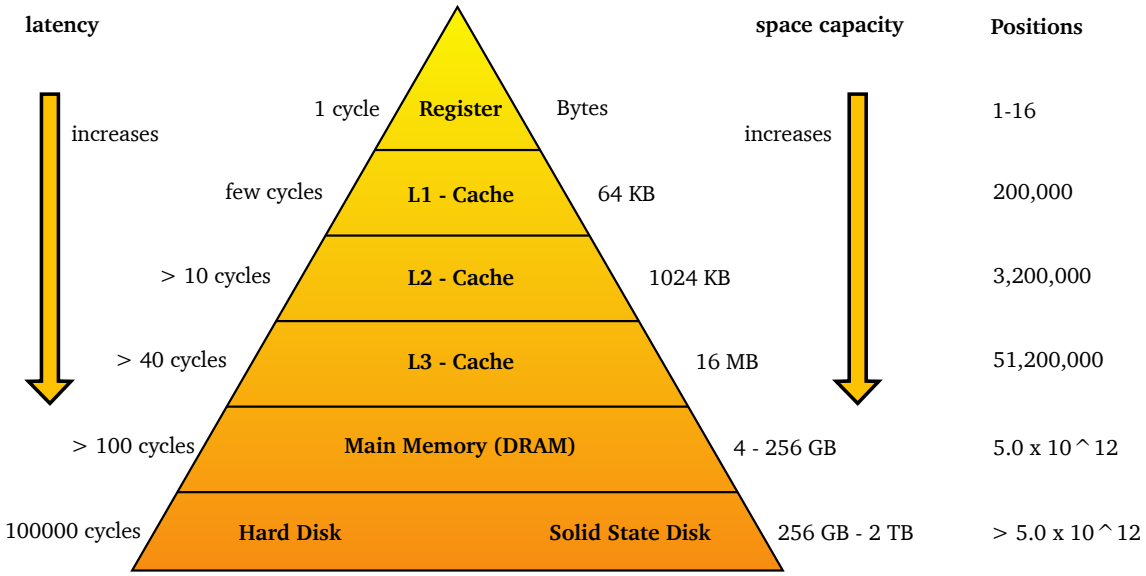


Figure 5.5: Storage hierarchy of actual consumer hardware. Unpredictable accesses to the main memory and especially to the Hard Disk are very time intensive. Thus, these so called cache misses should be avoided. The right most column expresses a rough estimation of positions which fit into the given space without Huffman encoding.

to reduce the number of lookups between several sub-database. However in our opinion it is really hard to find a way to fully avoid slow lookups into other sub-databases.

In conclusion, given actual hardware it is impossible to solve Pentago in less than a few years by using the Retrograde Analysis. Nevertheless the further development especially of the SSDs might change this fact. The rapidly increasing number of random access onto different blocks in comparison to HDs [10, 12] increases the chance to handle the dependency issues between the different sub-databases.

Time approximation: To underline our doubts, we make an optimistic calculation about the required amount of time to solve Pentago by Retrograde Analysis. Our simple approximation is based on three factors. At first we estimate the costs for fetching a position, calculating all successors or predecessors and store back the result with 1000 instructions. Second we assume that every 10th position requires a lookup in the main memory which is estimated with another 100 instructions. Last we assume that every 1000th state requires a hard disc access with the cost of around 10,000 instructions. As prior in this chpater, our estimation of the required time is based on the performance of an actual Intel Core-i7:

$$
\begin{aligned}
n &= 3.5 \cdot 10^{16} \\
instructions_{pos} &= 1000 \\
instructions_{mm} &= 100 \\
instructions_{hd} &= 10,000 \\
instructions_{total} &= n \cdot instructions_{pos} + \frac{n}{10} \cdot instructions_{mm} + \frac{n}{1000} \cdot instructions_{hd} \\
ips_{i7} &= 146,000 MIPS \\
time &= \frac{instructions_{total}}{ips_{i7} \cdot 10^6} \\
&= 2,660,958,904 \,\text{seconds} = 84 \text{ years}
\end{aligned}
$$

So optimistically speaking, in total 84 years are required to solve Pentago by utilizing the Retrograde Analysis. Only linear scaling in parallel execution will make a pure Retrograde Analysis feasible with a reasonable number (<100) of computers. But a linear scaling would require perfect decomposition, which is as explained not given. Although these are all very optimistic assumptions with the further hardware development a solving process might become realizable within the next years.

5.5 Hybrid approaches

Forward and backward search can be combined to a *meet-in-the-middle* or *hybrid* approach. The key idea behind this approach is to build up an endgame database with the Retrograde Analysis and to find a way from the initial node to the endgame database. Gasser[16] was able to solve Nine Men's Morris by using Alpha-Beta Search and endgame databases. Schaeffer solved *checkers* in 2007 with a hybrid approach [30] by using Proof-Number Search and Alpha-Beta Search in combination with a large Retrograde Analysis. Further means to solve *checkers*, where an initial line of play to set a strong seed for the Search algorithms. Figure 5.6 illustrates such a generic hybrid approach to solve games.

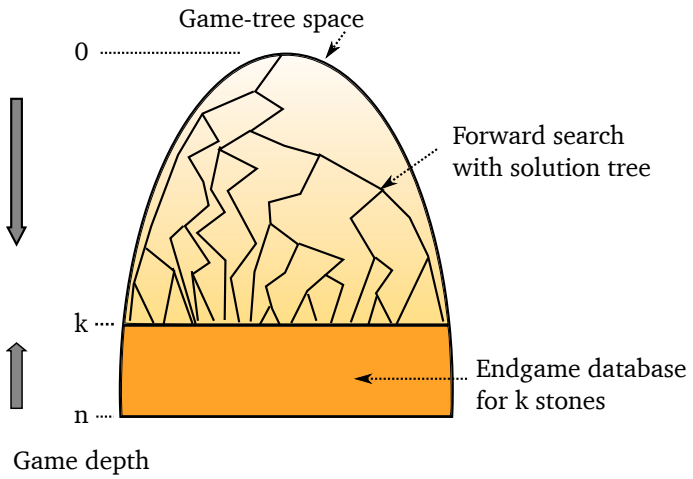


Figure 5.6: Hybrid search A endgame database up to k stones is created via Retrograde Analysis. Then a forward tree search tries to hit into the endgame database.

As mentioned Schaeffer et al. also combined the Proof-Number Search with the Alpha-Beta Search to solve *checkers*. They used a PN algorithm to maintain the whole game-tree and an efficient Alpha-Beta Search implementation as a second level search. The so called *Proof Manager* controls the direction of the search and distributes positions to solve by the second level search algorithm. If the Alpha-Beta Search cannot prove the given position, either a second level PN Search is initiated or the Proof Manager expands the node to further divide the task.

Applicability onto Pentago

The first mentioned hybrid approach is confronted with the same problems as the Retrograde Analysis. Looking at Table 4.1 in Chapter 4, we see that that only the last six depths in the state-space are smaller than one order of magnitude than the biggest part of the database. Since Pentago is a divergent game, the state-space hardly shrinks with increasing depth. Another reason that opposes the endgame database approach is, that our estimations of the game-tree size showed, that typical games do not reach the full depth or even a 6-stones endgame database. Thus in conclusion either the whole game can be solved by using a Retrograde Analysis, or given actual hardware it is even pointless to construct an endgame database.

The second combination of a Proof-Number Search with an Alpha-Beta Search seems more promising to use. The size of the game-tree might be to large to fit into the main-memory for the PN Search. But a smaller fraction beginning at the root might fit inside the main-memory and an Alpha-Beta Search

Table 5.2: Comparison of the presented algorithms, along with an approximation of the computational cost. The time estimation is calculated for a single CPU.

Algorithm	Advantages	Disadvantages	Required time
Alpha-Beta Search	- easy to implement - founded knowledge base - can be used as AI	- game-tree too large → will not finish	- CPU bound - approx. 10^8 years
Proof-Number Search	- optimized and developed for solving - profits from unbalanced game-tree	- state-space is too large to fit in memory	- Memory bound - might be faster than $\alpha\beta$
Threat-based Search	- highly optimized for solving go-moku - reduces problem size	- does not suit onto Pentago - finding threats is too costly - many counter moves possible → no parallel execution	- not applicable
Retrograde Analysis	- easy to implement - operates only on state space	- state space might be too large - rotating violates storage locality	- memory bound >84 years

algorithm might solve the leafs of the PN tree. However these speculations require further testing and evaluation to give a strong statement.

5.6 Conclusions

Table 5.2 sums up all mentioned algorithms next to their advantages and disadvantages regarding an application onto Pentago. The Retrograde Analysis has the lowest approximation of required time but it is useless to partly construct an endgame database which cannot be used any further. Thus, we decided to begin with an implementation of an Alpha-Beta Search algorithm, although a Proof-Number Search approach might require less computational time. But one reason convinced us to start with an Alpha-Beta Search implementation, since it still serves as a game AI, even when it is not possible to solve Pentago. Furthermore, with the results of an Alpha-Beta Search implementation, better approximations of the game-tree size and complexity are possible. Details of the implementation are discussed in the next chapter.

6 PentagoAI

As seen before, all discussed solving algorithms have drawbacks. We decided to begin with an Alpha-Beta Search implementation which still can serve as a Pentago AI when the solving process reveals as impossible. The components of our program PentagoAI are discussed in the following sections. We begin with a discussion about already existing programs that play Pentago in Section 6.1. Followed by an overlook on the basic structures in PentagoAI in Section 6.2. Section 6.3 discusses the implementation details of the Alpha-Beta Search algorithm. Finally the last Section 6.4 describes our observations with PentagoAI.

6.1 Related AIs

As of our knowledge, there exist only two serious open source artificial intelligence for Pentago named *pentagod* written by Tewalds [39] and *Pentago Ag* written by latkin [22]. The programs name of *pentagod* derives from its ability to run as a httpd daemon for its graphical user interface (GUI). Tewalds implements several algorithms in C++ to play Pentago like fixed depth Negamax, Negascout and UCT. He describes his Alpha-Beta Search implementation with: ”As far as I know, the negamax version is the strongest pentago player in existence right now. “. Hence his program seems to be a well suited opponent for testing purposes. Also at least two Java Pentago implementation exists. But a deeper look into their source codes points out, that their Alpha-Beta Search algorithm is not optimized in any way. Furthermore as a human amateur it is quite easy to win against both AIs.
Lim Chee Aun [6] developed an animated Javascript interface named *pentagoo* in combination with a simple php AI backend. He describes his AI as a weak test implementation. However the main component is the frontend which is really well organized and easy to use along nice graphic components. Since *pentagoo* and *pentagod* are released under the open source MIT license we hook into *pentagod*, which further utilizes *pentagoo* to make use of the well developed http interface. Further details are discussed in Section 6.3.

Also a few commercial products for smartphone devices are available. We took a short look at the Symbian OS version [25] but we could only played testgames, since the lack of any source code. The results of the comparison between both programs is shown in Section 6.3.

6.2 Implementation overview

Our program *PentagoAI* was developed in C++. Although we avoid object orientated programming to maintain the best control over all storage structures and optimizations, we profit from the simpler I/O concepts in C++. As described earlier, our program implements the *pentagod* httpd code as a frontend for human versus AI games. Our benchmarks run without the httpd code and print all output either to console or to file. Moreover the library *Boost C++* [8] is used to read parameters from arguments or configuration file. The advantage of the *Boost C++* API is a really easy way to access configurations files.
Our program is based on an iterative deepening Negamax search algorithm. We did not mention the algorithm before, but it is in principal an Alpha-Beta algorithm which combines the two different functions for the MAX and the MIN node to maximize the outcome for both players. Therefor only one function is called with inverted alpha and beta value. A pseudo-code Negamax is given in Algorithm 2 in the Appendix. Typically the Alpha-Beta Search function is (recursively) called with five parameters:

alphaBeta(node, depth, alpha, beta, player), which requires a branching to handle the difference between MAX and MIN nodes. Algorithm 3, also in the Appendix, shows this in detail. Now to avoid this branch, which hardly has an impact on the complexity, at every depth the same function is called but the alpha and beta values are exchanged and inverted. Furthermore the evaluation function outputs higher values if an advantage for the actual player is calculated and lower or negative values for a disadvantage. In contrast, the former evaluation function returned turn independent high values if an advantage for the MAX player existed.

Launching and playing PentagoAI

In principal PentagoAI is invoked with "./PentagoAI␣PARAM". Whereas the Parameters may also be placed in a file named "pentagoAI.cfg" in the same directory of the binary. Here we only describe the main options but the help of PentagoAI is in a sense self explaining.

Help: To display all parameters the program has to be called with "./PentagoAI␣−−help".

GUI: To run the program with the GUI interface on top of the httpd Daemon: "./PentagoAI␣−d". Further httpd options are the hostname/IP (-h) and port(-p) to listen on. Moreover the *pentagoo* dir(-l), where the *pentagoo* html files are located, may be specified.
It is also possible to configure the AI by setting either a fixed depth ("−−negamax_depth␣N") or a time limit ("−−negamax_id_time␣T") for the iterative deepening search.

Benchmark: To Benchmark the performance of PentagoAI, the program has to called with "./PentagoAI␣−b". The default benchmark initializes two iterative deepening PentagoAIs to challenge each other in a single match. To increase the number of played games the parameter "−−benchmark_games␣N" has to be added. To run the benchmark from random start positions instead of the initial one, "−−benchmark_random_starts␣true" must be enabled. Furthermore it is possible to use the *pentagod* player as opponent ("−−opponent_pentagod␣true"). The search depth of the PentagoAI player may also be configured, see "−−help" for more details.

The http daemon listens per default on ”http://127.0.0.1:9999/“ and may be accessed through an Internet browser to play Pentago. The browser should display *pentagoo* with its interface shown Figure 6.1.

The game pops up with a human versus human match, which may be changed to human vs. computer by choosing *New Game* → *Player 2* = *Computer* → *Start Game*. The selection of the AI strength is currently not implemented. Figure 6.2 shows a screen-shot of the settings menu.

6.3 Implementation details

This section is divided into six parts. We begin with an introduction of the used structures and BitBoard technique. Then the details of the move generation are discussed, followed by a description of the evaluation function. After that, the iterative deepening implementation is explained. We will finish with the details of the used Transposition Table and implemented move ordering.

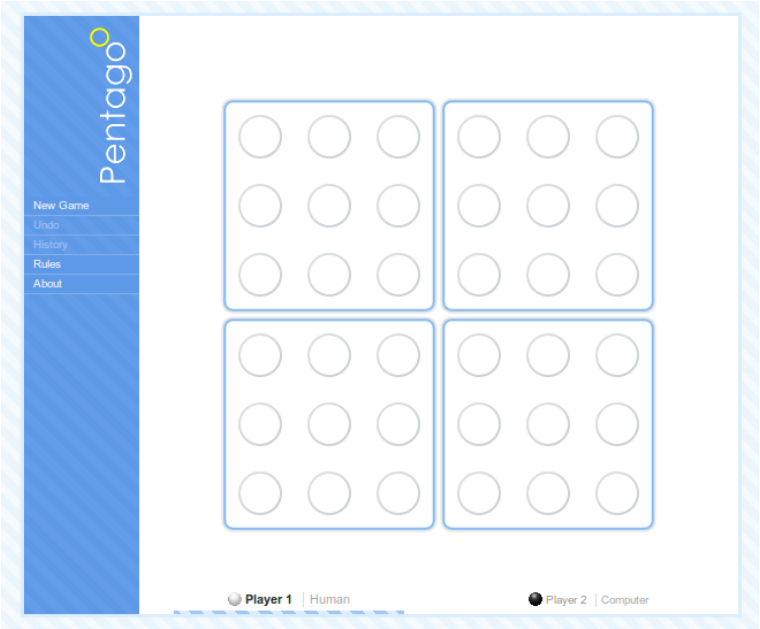


Figure 6.1: The frontend of PentagoAI, using *pentagoo*.

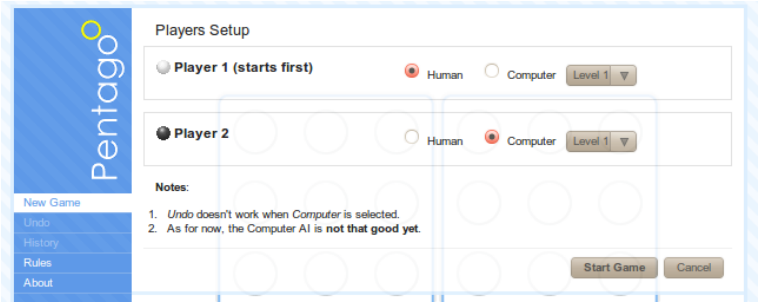


Figure 6.2: The *New Game* menu of *pentagoo*.

6.3.1 BitBoards

A gamestate in PentagoAI consists of two components, a board description and a move counter. The move counter contains redundant information, since it is possible to derive its value from the boards state. Thus it may be deleted, but we maintain the counter for performance reasons.
In general there are two ways to map a board to computational units. The naive approach is to construct an array with 6x6 entries where each entry may be set to one of the three possible values (empty, occupied by black, occupied by white). This requires at least 36 Bytes, if the variable type Byte is supported by the compiler. Typically accesses on aligned datatypes like Integers are quite faster than single Byte accesses. Using Integers this results in 36 Integers per board, which require architecture depending either 144 Bytes (32 Bit) or 288 Bytes (64 Bit). Despite the fact, that 36 Integers do not fit into registers of typical x86 hardware, the evaluation function requires more instructions to count all occurrences of k-in-a-row. It is quite a performance advantage if the whole board fits into registers to reach the maximum possible computational power instead of fetching and pushing positions from and to the cache. To achieve this goal we used BitBoards. We maintain two 64-bit Integers (type *uint64_t* in C++), one for each color. Every field on the board is mapped to a single bit inside the Integer, beginning at the lowest significant bit. Figure 6.3 shows the whole mapping along with an example

board configuration. The mapping is the same for both colors. Hence if a white marble is set on position

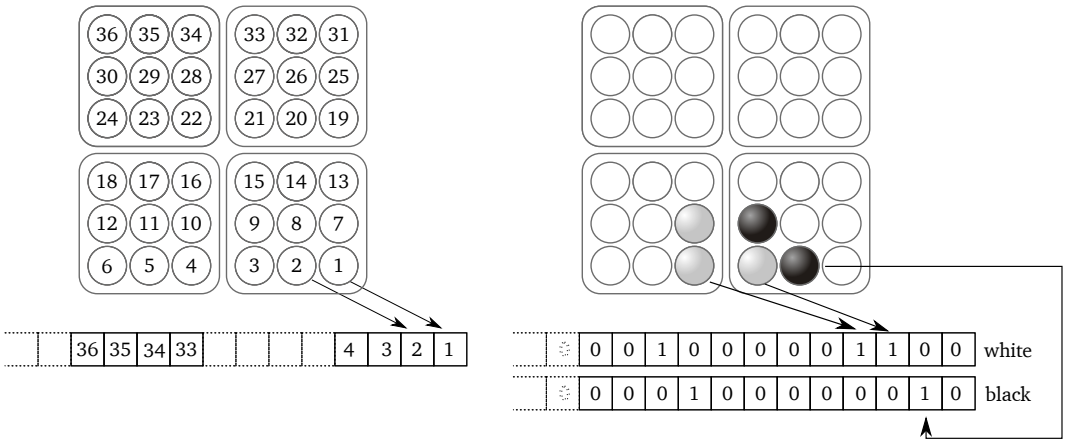


Figure 6.3: Linear mapping of each position in the board onto the bit string.

15, the 15th bit in $bitboard_{white}$ is set to 1. The logical AND of both BitBoards always returns 0 since only one marble per field is allowed. We chose a two Integer representation because two times 36 bit does not fit into a single 64 Bit Integer.
Summing up, a gamestate is presented by the following structure in Listing 6.1:

Listing 6.1: The Pentago BitBoard structure $P\_BITBOARD$. The type $BITBOARD\_T$ is a 64 Bit Integer.

```
1 typedef struct {
2     BITBOARD_T board[2];
3     unsigned int move;
4 } P_BITBOARD;
```

To give an example how the BitBoards are accessed, Listing 6.2 shows an example of the *setPos()* function which places a marble on a given position. It is easy to see that only two more but fast instructions are required to access a chosen bit. These are the load immediate to push the 1 into a register and the left shift. The logical OR just replaces a move instruction which would be required in an equal array representation to set an element.

Listing 6.2: The *setPos()* functions places a marble onto the board given position and color.

```
1 inline void setPos(P_BITBOARD *b, int pos, int color)
2 {
3     b->board[color%2] |= (BITBOARD_T)( 1L << (pos));
4 }
```

6.3.2 Move generation and execution

The move structure consists of three parameters: The position where to set a new marble, the quadrant which has to be rotated and the rotation direction. As described before, to place a marble to a position *pos* a single bit in a bitmask is set and connected with a logical OR to the boards BitBoard. The rotation of the quadrants is the bottleneck of the move execution. Every single bit of the affected quadrant has to be fetched, shifted and pushed back in a temporary copy of the BitBoard. Because eight fields have to be rotated this results in a very costly operation, thinking of billions generated moves. It might be possible that better representation structures exist but they will all have a drawback on the simplicity of

the evaluation function.

The $getMoves()$ functions returns a linked list with all possible moves for a given position. The function iterates over all fields onto the given board and checks whether they are empty or not. If an empty field is found, all eight possible rotations of the quadrants are appended to the linked list together with the position where to place the marble.

6.3.3 Negamax Search algorithm

The practical Negamax implementation is almost the same as the pseudocode description 2 in the Appendix. Only the pulling and storing of evaluation values is added. Therefor every evaluated position is added to the Transposition Table. Furthermore directly after $isTerminal()$- the $lookup()$- function is called. This avoids needless examination of children.

6.3.4 Evaluation function

An evaluation function rates a given position according a heuristic. Thus it tries to figure out, which player has a more likely chance to win the position. As extensively described earlier, it is really hard to give a reliable heuristic for Pentago because each rotation of a quadrant may totally change the situation. Hence we decided, to reduce the complexity of the evaluation function to extend the search depth to a deeper level, this is a common approach [28].

The main component of our evaluation function check for a winning position. This is done by using a bitmask and a logical AND. Two instructions are sufficient to check for a five-in-a-row: "if (bitmask&b->board[color])==bitmask))", if we assume that the mask and the board are already in memory. Each further check of five-in-a-row requires one extra shift of the bitmask to iterate over all positions. To give examples the horizontal bitmask is $bitmask_{horizontal} = 31 = 011111_b$, whereas the vertical bitmask is given by

$$\begin{aligned} bitmask_{vertical} =& 820820800_h \\ =& 100000 \\ & 100000 \\ & 100000 \\ & 100000 \\ & 100000 \\ & 000000_b \end{aligned}$$

The diagonal bitmask is build in the same way. Now each bitmask is wrapped by a loop to shift it over every possible occurrence of a five-in-a-row. In conclusion, it is possible to search over the board with around 100 instructions per side. This concept may be transferred to check for four or even three in a row, too. It is important to note, that Pentago demands a check of the whole board for both players after every move, not only local around the last placed marble since the rotation produces new game situations.

The second component is based on a simple heuristic to achieve a better gameplay: Marbles next to

center of the board get a higher rating than marbles touching the boards margin. We observed, that midpoint marbles may join more k-in-a-rows possibilities than marbles next to the border. However the rating for good placed marbles is far below the rating for a three-in-a-row and even lower for the five-in-a-row. An example rating is given in Table 6.1. We implemented a linear evaluation function:

Table 6.1: Weights used in the linear evaluation function.

Occurrence x_i	**Weight** w_i
5-in-a-row	100,000
4-in-a-row	1,000
3-in-a-row	100
marble in center	5
marble at board	0

$$eval(pos) = \sum w_i * x_i$$

Hence both components are summed up according the given weights for both colors. Finally the difference between the actual player and his opponent is calculated.

6.3.5 Transposition Table

The Transposition Table is realized in two different implementations. On the one hand the Standard Template Library (STL) hashmap, provided with C++, is used, on the other hand the slower google-sparshash hashmap[18]. The letter one is more space-efficient which allows larger Transposition Tables. We do not need considered any replacement schemes because we utilized well implemented hashmaps. To avoid memory issues, the Transposition Table is only used from depth 1 to $d - k$ whereas k has to be specified by the user. The reason for this decision is that the deeper depths may be as fast calculated as looked up inside the Transpositions Table, especially the level of the leafs. The performance of the different Transposition Tables and techniques is discussed in the next section.

6.3.6 Move ordering

As extensively described earlier move ordering is very important for a good performance of the Alpha-Beta Search. We have only two means to order the moves, since the lack of any good reliable heuristic. The $getMoves()$ function has to execute every move and if a resulting position reveals as a win for the actual player, the position becomes the new head of the linked list. Furthermore the resulting position is checked against the Transposition Table, all positive rated positions are placed directly after the winning positions. All other positions are added to the end of the list in order of their occurrence. The linked list perfectly suits our needs for an ordered list of moves. Fast adding of positions is played out via pointers to the head and tail of the victory positions.
This procedure might be cost intensive but as shown earlier a good move ordering may half the search depth.

6.4 Observations and benchmarks

Given the program PentagoAI we had three different possibilities to benchmark and observe its performance. At first we could play ourselves against PentagoAI, second we could let it play against itself and last we connected pentagod with PentagoAI to test the performance against another AI.

6.4.1 Human versus PentagoAI

We begin with our non-quantitative observations during playing against PentagoAI. We presume that our Pentago skills are on an advanced amateur level. However the first thing we noticed is that the program starts totally undirected into the game. The marbles are set on heuristically good rated fields but no consequent way to five-in-a-row is visible. This behavior was expectable. since we do not use any opening books. Although no perfect way to win was visible to us, we never won or played a draw against PentagoAI, despite some crashes due to programming failures. Even as the first player, concentrating only on defensive moves, we had no chance. So a match developed rather slow and randomly but ended with a loss on our side. Now, what are the strengths of PentagoAI? The exact search depth is discussed in the next paragraphs, but typically PentagoAI is able to search at least 4 ply. This is enough to defend any occurring open three- or four-in-a-row. Furthermore after a few marbles are placed, as a human amateur you have to strongly focus not to oversee any loosing position. Its is unlikely harder to see more than 1 ply in forecast than in other typically board games. This is probably because of two reasons. At first their is a large branching factor in every move, which is still smaller than for example Go. The other fact is, that the positions may totally change for every move. This seems to make the game much more complicated for a human player. In result without any > 6 ply winning move sequence or strategy, we do not see any chance for us to win.

6.4.2 PentagoAI versus PentagoAI

We used the possibility to observe matches between PentagoAI and itself to tune different parameters. Furthermore the average game depth $depthavg = 21.03$ was monitored in 200 games of PentaogAI with the search depth of four. In average the time to evaluate a position with the search depth of four is below ten seconds. But each further step in the search depth requires at least the factor around 10-20. Hence an evaluation of the search depth of five finishes in around one minute. This shows, that PentagoAI has no chance to solve Pentago in a reasonable amount of time.
Moreover we did not any sing of a non-uniform tree, which might have reduce the game-tree complexity. Looking at the evaluation function, small changes in the weights of the heuristic did not show any significant influence on the performance. Even more interesting to see are the consequences when one or more parts (four-in-a-row, three-in-a-row etc.) of the heuristic are removed. We observed that the strengths of PentagoAI decreases with a less complex evaluation function. However the runtime also decreases, but not far enough to reach a deeper search depth under the same time constraints. Hence we decided to use the complex evaluation function for all further benchmarks.
We also measured the required time to evaluate a position with the two different hashmaps mentioned above which are used as Transpositions Tables. They both handle collisions themselves therefor they do not change the outcome of the Alpha-Beta search. Collisions may occur if two or more positions map on the same entry in the hashmap and this case has to be handled, else a lookup in the Transposition Table would read a value for a different position. Our observations showed, that the STL hashmap is slightly faster than the googe-sparsehash hashmap. But the latter one requires far less memory, which would be interesting for a solving approach. All following benchmarks are based on the STL hashmap.

6.4.3 Pentagod and others versus PentagoAI

As said, there exist only a few AIs for Pentago, but *pentagod* and *Pentago Ag* [22] seem to be the only comparable opponents. Nevertheless we also took a short look at *Pentago for Symbian OS* [25] and *Pen-*

tagoPP [45].

PentagoPP: *PentagoPP* is written in Java and was developed in a course at the Georg-Augustin-University in Germany (Göttingen). Its main focus lies on its network component and on the visualization of the board but they also included a basic AI. Unfortunately testgames with *PentagoPP* showed, that the search depth is too low to stand against *PentagoAI*.

Pentago for Symbian OS: There a similar versions for the Iphone OS available, but we took a look at the closed source Symbian version. Since we are unable to connect both AIs we had to enter the moves by hand. As expected due to the hardware limitations *Pentago for Symbian OS* lost all ten played games against *PentagoAI*. Probably the strength of this commercial product is limited by purpose to address more customers.

pentagod: As mentioned, we compared *PentagoAI* with pentagod which is described by the author with "As far as I know, the negamax version is the strongest pentago player in existence right now."[39]. *Pentagod* implements a k-ply fixed depth Alpha-Beta Search without Transposition Tables. The evaluation function is very similar to the one of *PentagoAIs*. All three, four and five-in-a-row are counted, linear weighted and summed up to a heuristic value. BitBoard techniques are not used, hence the accesses to single fields on the board are realized with to array accesses.
We begin our comparison with a setup of a fixed depth search on both sides. The results are presented in Table 6.2. The first color to move is white. Twenty matches are played, where each AI plays 10 times as white and another 10 times as black. It is good to see that both algorithms perform nearly on the same

Table 6.2: Comparison of *Pentagod* and *PentagoAI* by search depth. Twenty games were played, where each AI plays ten times as white.

pentagod **PentagoAI**	**3-ply** **3-ply**	**3-ply** **4-ply**	**4-ply** **3-ply**	**4-ply** **4-ply**
Win for pentagod	11	4	18	5
Win for PentagoAI	9	15	2	8
Draw	0	1	0	6
Win for White	11	12	10	7
Win for Black	9	7	10	7
Total	20	20	20	20

level but *pentagod* has a slightly advantage with the same search depth. This is probably based on the different evaluation strategies. *Pentagod* evaluates every three-in-a-row, whereas *PentagoAI* only evaluates four- and five-in-a-row. But as expected, an increased search depth by one ply results in a advantage for the respective AI. However these results are more or less meaningless without a comparison of the required computational time. Hence we decided to compare the runtime of both programs to evaluate a given position for a specified depth. Both programs run on the same hardware (Intel Core2Duo T7700 2,4Ghz; 4GB of Main Memory) with the same compiler setup (gcc-4.3.3 x64 -O2). The results are shown in Table 6.3.

Comparing the runtimes, it is good to see that *PentagoAI* is able expand more nodes per second than *pentagod*. But it is also very interesting, that although BitBoard techniques and Transposition Tables are used both programs only differ by such a low factor. The performance difference between *pentagod*

Table 6.3: Comparison of *Pentagods* and *PentagoAIs* runtime to evaluate a position for a given depth. The initial position is the empty board, the other positions are generated by random.

		pentagod		**PentagoAI**	
	depth	time [ms]	pos / second	time [ms]	pos / second
initial position	3	10	2959833 ply/s	76 ms	3840597 ply/s
initial position	4	79	3134635 ply/s	775 ms	5420301 ply/s
random (7 marbles)	3	66 ms	3219924 ply/s	88 ms	4132180 ply/s
random (7 marbles)	4	4192 ms	3453730 ply/s	1212 ms	6036505 ply/s
random (12 marbles)	3	95 ms	2734171 ply/s	170 ms	4470351 ply/s
random (12 marbles)	4	2307 ms	3151974 ply/s	1200 ms	6685262 ply/s
random (20 marbles)	3	105 ms	3547428 ply/s	112 ms	3745221 ply/s
random (20 marbles)	4	726 ms	3489536 ply/s	477 ms	7140927 ply/s

and *PentagoAI* rises with after the first marbles are placed on the board. This is because *pentagod* discards symmetric positions during move generation. Since symmetric positions only occur during the first moves which are more or less played randomly we omitted the removal of those. But it is definitive useful to implement this feature in a future version of *PentagoAI*.
Unfortunately the time advantage for *PentagoAI* is not large enough to evaluate one more depth. Thus, both programs are on the same level, *pentagod* convinces with its evaluation function, but is also slower than *PentagoAI*. In contrast to *PentagoAI*, *pentagod* is able to support multithreading on the first level of the Negamax Search. This decrease the evaluation time given a multi core system. Since we did not yet implement multithreading we did not take the multithreaded version into account.

Pentago Ag: This Pentago AI is developed in C# and its algorithmic base is quite comparable to *pentagod* and *PentagoAI*. *Pentago Ag* makes use of BitBoard techniques but does not implement Transposition Tables. Since it is written in C# we did not spend the time to play a tournament with *Pentago Ag*. However the author states on his homepage [22] that *Pentago Ag* is able to evaluate $550,000$ - $600,000$ position per second on an 2,4GHz x64 processor, which is the same as the authors used for *PentagoAI*. Moreover a short look in the source code of *Pentago Ag* reveals that the number of evaluated position is counted in the same way as in *pentagod* and *PentagoAI*. Hence we strongly assume, that *PentagoAI* is quite stronger than *Pentago Ag*.

7 Conclusions

Our first look at the solveability of Pentago seemed very promising but during the work of this thesis many hard and complex problems occurred. Probably the relatively small board size of $6\,x\,6$ was very misleading.
We approximated the runtime of four different solving approaches but non of them would finish in reasonable amount of time. The Alpha-Beta and Proof-Number Search are confronted with a too large game-tree and a threat based search algorithm is not easily applicable. A Retrograde Analysis requires better decomposition schemes to succeed on todays hardware.
Now we can only speculate about the game-theoretical outcome in Pentago. However we can apply the *Strategy Stealing Argument* [14] which states that if the second player would know a strategy to win than the first player could apply this strategy for himself. This argument only holds for symmetric boards where both players have the same possibilities to move with the same outcome which is given in Pentago. Hence Pentago cannot be a game theoretic win for the second player. But our observations with PentagoAI in Chapter 6 showed that no clear advantage for the first player is visible. This leads us to the guess that Pentago might be a game theoretic draw. However this is a very rough estimation since it is based on the naive Alpha-Beta Search implementation of PentagoAI.

Despite the fact, that we were not even close at solving Pentago a few interesting observations were made. The simple Alpha-Beta Search implementation of PentagoAI is able to beat us human players although it reaches only a rather small search depth. This is a very interesting and surprising fact since this opposes many other board games already researched in the field of AI. To give an example, the branching factor at the beginning of Pentago is comparable to the branching factor of *Go*, where in contrast actual programs are still on an amateur level. Another positive outcome of this thesis is that PentagoAI is one of the best AIs for Pentago. We cannot offer any proof but we won the most games against all our known opponents in a fair comparable setup.

We left an open field for future work on Pentago. On the one side, we disregarded knowledge based methods like Pattern Search and on the other side our Artificial Intelligence implementation is by far not complete. Many advances in Alpha-Beta Search, like Aspiration Window and better Transposition Tables and replacement schemes could be applied. Also a strong UCT approach would be interesting to see. Furthermore in our opinion Pentago is still an interesting game and probably a Retrograde Analysis will become feasible within the next ten years.

8 Acknowledgments

I want to thank family and friends, particularly Peter Heise, Benjamin Milde, Rober Müller and Katharina Werth for taking the time to read my thesis and for the really motivational comments. Furthermore I want to thank Felix Büscher for introducing me into vector graphics.
Also, I acknowledge gratefully Prof. Dr. Johannes Fürnkranz for his supervision of my thesis. I especially enjoyed the freedom he gave me in my research in the interesting field of Artificial Intelligence.

9 Appendix

Algorithm 2: Negamax Search algorithm. Negamax maximizes the outcome for both players. The $evaluation()$ function returns higher values if the actual player has an advantage. The output of the best move is omitted.

Input: A board configuration(position), the search depth and a lower alpha and an upper beta border.

Output: The evaluation value of the minimax tree.

```
Input: position, depth, alpha, beta
Output: bestValue
if isTerminal(position) or depth=0 then
    return evaluate(position)
end
bestvalue ← −∞
moves ← generateMoves(position)
foreach m in moves do
    successor ← doMove(position, m)
    value ← −negaMax(successor, depth − 1, −beta, −alpha)
    bestvalue ← max(value, bestvalue)
    if bestvalue >= beta then
        return beta
    end
    if value > alpha then
        alpha ← bestvalue
    end
end
return bestvalue
```

Algorithm 3: Alpha-Beta Search algorithm. The output of the best move is omitted. The variable *player* changes by turn between MIN and MAX. The *evaluation*() function returns higher values for positions with an advantage for the first player and lower values for a disadvantage.
Input: A board configuration(position), the search depth,a lower alpha, an upper beta border and the player to move.
Output: The evaluation value of the minimax tree.

```
Input: position, depth, alpha, beta, player
Output: bestValue
if isTerminal(position) or depth=0 then
    return evaluate(position)
end
moves ← generateMoves(position)
if player = MAX then
    foreach m in moves do
        successor ← doMove(position, m)
        value ← alphaBeta(successor, depth − 1, alpha, beta, MIN)
        if value > beta then
            return beta
        end
        if value > alpha then
            alpha ← value
        end
    end
    return alpha
else
    foreach m in moves do
        successor ← doMove(position, m)
        value ← alphaBeta(successor, depth − 1, alpha, beta, MAX)
        if value <= alpha then
            return alpha
        end
        if value < beta then
            beta ← value
        end
    end
    return beta
end
```

References

[1] JD Allen. A note on the computer solution of Connect-Four. *Heuristic Programming in Artificial Intelligence*, 1989. 3.1, 3.2, 4.3

[2] LV Allis. A knowledge-based approach of connect-four. *Master Thesis*, 1988. 3.1, 3.2, 4.3

[3] LV Allis. Qubic solved again. *Heuristic Programming in Artificial Intelligence*, 1992. 3.1, 4.3

[4] LV Allis. *Searching for solutions in games and artificial intelligence*. PhD thesis, 1994. 3.1, 3.2, 4.3, 5.2, 5.3, 5.3

[5] Thomas Anantharaman, Murray S Campbell, and Feng-hsiung Hsu. Singular extensions : Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990. ISSN 0004-3702. doi: DOI:10.1016/0004-3702(90)90073-9. 5.1.1

[6] Lim Chee Aun. Pentagoo - Pentago game in Javascript, 2009. URL http://code.google.com/p/pentagoo/. 6.1

[7] J. Bal and H. Romein. Solving the game of awari using parallel retrograde analysis. *IEEE computer*, 36, 2003. 3.2, 5.4.2

[8] Boost. Boost C++ Library, 2011. URL http://www.boost.org/. 6.2

[9] DM Breuker. Replacement Schemes and Two-Level Tables. *ICCA Journal*, 19:175—-180, 1996. 5.1.1

[10] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. 5.4.3

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms, third edition*. MIT Press, 2009. 5.4.2

[12] Cagdas Dirik and Bruce Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. *ACM SIGARCH Computer Architecture News*, 37(3):279, June 2009. ISSN 01635964. doi: 10.1145/1555815.1555790. 5.4.3

[13] Stefan Edelkamp and Damian Sulewski. GPU Exploration of Two-Player Games with Perfect Hash Functions. *Third Annual Symposium on Combinatorial Search*, AAAI:23–30, 2010. 5.4.3

[14] Richard K. Guy Elwyn R. Berlekamp, John Horton Conway. *Winning ways for your mathematical plays, Volume 3*. Wellesley, Massachusetts: A. K. Peters Ltd., 2nd editio edition, 2004. ISBN ISBN 1-56881-143-8. 7

[15] H Finnsson. Cadia-player: A general game playing agent. *Master's thesis, Reykjavik University*, 2007. 2.2

[16] R Gasser. Solving nine men's Morris. *Computational Intelligence*, 1996. 3.1, 3.2, 4.3, 5.5

[17] Ralph Gasser and Informatik Eth. Applying retrograde analysis to nine men's morris. *Heuristic Programming in Artificial Intelligence 2: the second computer olympiad*, pages 161–173, 1991. 5.4.2

[18] Google Inc. google-sparsehash - An extremely memory-efficient hash_map implementation, 2011. URL http://code.google.com/p/google-sparsehash/. 6.3.5

[19] M Heule and L Rothkrantz. Solving games - Dependence of applicable solving procedures. *Science of Computer Programming*, 67(1):105–124, June 2007. ISSN 01676423. doi: 10.1016/j.scico.2007.01.004. 3.1, 4.3, 4.4, 5, 5.3, 5.3

[20] D Knuth. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. ISSN 00043702. doi: 10.1016/0004-3702(75)90019-3. 5.1, 5.1.2

[21] R. Lake, J. Schaeffer, and P. Lu. Solving large retrograde analysis problems using a network of workstations. *Advances in Computer Chess*, 7:135–162, 1994. 5.4.2, 5.4.3

[22] Latkin. PentagoAg - A C# Pentago AI, 2009. URL http://pentagoag.codeplex.com/. 6.1, 6.4.3, 6.4.3

[23] D McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, July 1988. ISSN 00043702. doi: 10.1016/0004-3702(88)90019-7. 5.2

[24] Mindtwister AB. Pentago Meisterschaft, 2010. URL http://www.pentago-meisterschaft.de/index.html. 2

[25] Nokia Inc. Pentago for Symbian, 2011. URL http://store.ovi.com/content/39814. 6.1, 6.4.3

[26] O Patashnik. Qubic: 4× 4× 4 tic-tac-toe. *Mathematics Magazine*, 1980. URL http://www.jstor.org/stable/2689613. 3.1, 4.3, 5.3

[27] Randall Munroe. xkcd Comic 832 - TicTacToe. URL http://xkcd.com/832/. 3.2

[28] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. February 2003. ISBN 0137903952. 5.1, 5.1.2, 6.3.4

[29] J Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, April 1990. ISSN 00043702. 5.2

[30] J Schaeffer. Game over: Black to play and draw in checkers. *ICGA Journal*, 2007. 3.1, 3.2, 4.2, 4.3, 5.5

[31] J. Schaeffer and R. Lake. Solving the game of checkers. *Games of no chance: combinatorial games at MSRI, 1994*, page 119, 1998. 3.1

[32] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992. 3.1

[33] J. Schaeffer, R. Lake, P. Lu, and M. Bryant. CHINOOK the world man-machine checkers champion. *AI Magazine*, 17(1):21, 1996. 3.1

[34] J. Schaeffer, Y. Bjornsson, N. Burch, R. Lake, P. Lu, S. Sutphen, and A. Edmonton. Building the Checkers 10-piece Endgame Database. In *Advances in computer games: many games, many challenges: proceedings of the ICGA/IFIP SG16 10th Advances in Computer Games Conference (ACG 10), November 24-27, 2003, Graz, Styria, Austria*, page 193. Springer Netherlands, 2004. 3.1, 5.4.2

[35] J. Schaeffer, N. Burch, Y. Bjornsson, A. Kishimoto, M. Muller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007. 3.1, 4.3, 5.1.3

[36] Jonathan Schaeffer and Aske Plaat. New advances in Alpha-Beta searching. In *Proceedings of the 1996 ACM 24th annual conference on Computer science*, CSC '96, pages 124–130, New York, NY, USA, 1996. ACM. ISBN 0-89791-828-2. 5.1.1

[37] Stephan Schiffel. Symmetry detection in general game playing. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 67–74, 2009. 2.2, 4.1

[38] Reza Shams, Hermann Kaindl, and Helmut Horacek. Using aspiration windows for minimax algorithms. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*, pages 192–197, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1-55860-160-0. 5.1.1

[39] Tewalds. pentagod - A pentago bot/ai, 2010. URL http://code.google.com/p/pentagod/. 6.1, 6.4.3

[40] Tom's Hardware. Review Core-i7-990x-Extreme-Edition-Gulftown, 2011. URL http://www.tomshardware.com/reviews/core-i7-990x-extreme-edition-gulftown,2874-6.html. 5.1.3

[41] Tony Warnock and Burton Wendroff. Search Tables in Computer Chess. *ICCA Journal*, 11(1), 1988. 5.1.1

[42] H. van den Herik and M. Winands. Proof-Number search and its variants. *Oppositional Concepts in Computational Intelligence*, Volume 155(Studies in Computational Intelligence):91–118, 2008. 5.2, 5.2.1, 5.2.2

[43] H. Jaap van den Herik, Jos W. H. M Uiterwijk, and Jack van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134:277 – 311, 2002. 3.1, 3.2, 4, 4.2, 4.3, 4.3, 5

[44] Wikipedia. Pentago — Wikipedia, The Free Encyclopedia, 2011. URL http://en.wikipedia.org/wiki/Pentago. 2

[45] Andreas Wilhelm. PentagoPP - A Java implementation of Pentago, 2008. URL http://www.avedo.net/pentago_pp/. 6.4.3