STUDIENGANG MATHEMATIK

DER FACHHOCHSCHULE GIESSEN-FRIEDBERG

Diplomarbeit

Das Minimax-Verfahren zur Variantenberechnung von Suchbäumen im Schach

Fritz Reul

1. Referent: **Prof. Dr. Annegret Hoy**
2. Referent: **Prof. Dr. Harmund Müller**

Inhaltsverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis und Gliederung

Im ersten Kapitel werden die Regeln und Grundlagen des Schachs vorgestellt. Die Figurenbewertung stammt aus der Software CRAFTY von *Robert M. Hyatt (1996-2001),* Associate Professor of Computer and Information Sciences, University of Alabama at Birmingham. Die Kriterien der Stellungsbewertung werden in dem „KURS SCHACHSTRATEGIE 1-3“ von *Alex Bartashnikov* vorgestellt.

Die Grundlagen der Suche sind sehr gut in dem Buch von *John White* „WRITING STRATEGY GAMES ON YOUR COMMODORE 64“ vorgestellt, und dienten in diesem Kapitel zusammen mit einem Artikel aus der Zeitschrift CSS 3/03 von *Stefan Zipproth* als Vorlage.

Die *Bitboard-Technologie* im zweiten Kapitel geht ebenfalls auf die Software CRAFTY von *Robert M. Hyatt* zurück. Zudem bilden die Artikel „BITBOARDS TEIL 1 UND 2“ von *Stefan Zipproth* aus den Zeitschriften CSS 4/02 und CSS 5/02 weitere Grundlagen für dieses Kapitel.

Die *Hashverfahren* im dritten Kapitel orientieren sich an dem Buch „SOFTWARETECHNIK IN C UND C++“ von *Professor Rolf Isernhagen*. Ausserdem wurden weitere Speichermodelle hergeleitet, die für die Variantensuche besonders effizient sind.

Das vierte und letzte Kapitel dieser Arbeit beschäftigt sich mit der *Variantensuche* und basiert besonders auf dem ersten und dritten Kapitel. Technische Erweiterungen des *Minimax-Verfahrens* gehen besonders auf die Arbeit von *Claude Shannon* zurück. Die algorithmischen Lösungen werden besonders gut in dem Buch „SCHACH AM PC“ von *Dr. C.Donninger* vorgestellt.

Die heuristischen Pruning-Verfahren wurden in dem Buch von *Ernst A. Heinz* genau untersucht, und sind in diesem Kapitel zusammen mit der Software CRAFTY von *Robert M. Hyatt* die Eckpfeiler. Als Abschluss dieses Kapitels wird das Minimax-Verfahren für die Variantensuche mit dem Minimax-Verfahren für *Matrixspiele* verglichen.

Vorwort

Schach ist perfekt spielbar, wenn alle möglichen Stellungen bekannt sind. Doch wieviele verschiedene Stellungen gibt es überhaupt? Und wieviele davon könnten im Rahmen einer normalen Schachpartie vorkommen? Fragen auf die bislang nur relativ ungenaue Antworten gegeben wurden.

Nach einer mathematischen Untersuchung von *Christoph Fieberg (CSS 2/02)* wurde die grobe Abschätzung aller denkbarer Stellungen von $13^{64} \approx 10^{72}$ (wegen 13 verschiedenen Materialverteilungen auf 64 möglichen Feldern) präzisiert. Die Summe der praxisrelevanten 3-Steiner bis 32-Steiner errechnet sich kombinatorisch und es ergeben sich nach *C. Fieberg* immer noch 3×10^{46} (30 Septillarden) Stellungen. Seit einigen Jahren liegen fast vollständige Daten aller Stellungen mit höchstens 6 Steinen auf dem Schachbrett vor, es sind dies bereits 4.3 Billionen Stellungen.

Es ist weder möglich sämtliche Stellungen zu berechnen, noch alle Stellungen, könnte man sie denn berechnen, auf einem Speichermedium aufzubewahren. Seit dem Vortrag von *Claude Shannon* am 9. März 1949 wird der Ansatz des Minimax-Verfahrens konsequent weiterverfolgt.

Die Gliederung dieser Arbeit ist so gewählt, dass das Minimax-Verfahren zur Variantenberechnung stückweise hergeleitet wird. Die bereits von *C. Shannon* beschriebenen Notwendigkeit bei der Variantensuche nur *ruhige Stellungen (quiescent positions)* zu bewerten wird in Kapitel 4 mit den bis dahin erarbeiteten Verfahren und mit der Hilfe moderner Algorithmen vorgestellt.

Die aus C-Code abgeleiteten Basisalgorithmen sind nur als Lösungsbeispiele gedacht. Ausserdem kann im Rahmen dieser Diplomarbeit nicht auf weitere Problemstellungen eingegangen werden, welche analog mit dem Minimax-Verfahren zu lösen sind. Aufgrund der Komplexität eines Variantenberechnungs-Algorithmus kann auch kein Wert auf Vollständigkeit der Beispielalgorithmen und Rechenbeispiele gelegt werden.

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Gründau, den 3. Dezember 2003

Fritz Reul

1 Einleitung

1.1 Das Schachspiel

1.1.1 Das Schachbrett

Schach wird auf den 64 abwechselnd weissen und schwarzen Quadratfeldern des Schachbretts von 2 Spielern mit je 16 Spielfiguren (je 1 König, 1 Dame, 2 Türme, 2 Läufer, 2 Springer, 8 Bauern) gespielt. Diese Spielfiguren dürfen nach jeweils eigenen Regeln abwechselnd mit je 1 Zug über das Spielfeld ziehen (Weiss stets zuerst).

```
  XABCDEFGHY
8rsnlwqkvlntr(
7zppzppzppzpp'
6-+-+-+-+&
5+-+-+-+-%
4-+-+-+-+$
3+-+-+-+-#
2PzPPzPPzPPzP"
1tRNvLQmKLsNR!
  xabcdefghy
```

Abbildung 1-1 Startstellung

1.1.2 Die Schachfiguren

- Die Bauern P rücken je 1 Feld geradlinig vorwärts, nur von der Grundlinie auch 2 Felder. Sie schlagen (besetzen das Feld einer gegnerischen Figur, die damit verloren ist) im Gegensatz zu den übrigen Figuren nur schräg.
- Die Springer N , die als einzige andere Figuren überspringen können, bewegen sich 2 Felder vorwärts und 1 Feld zur Seite in allen Richtungen.
- Die Läufer L bewegen sich diagonal, so dass sie nur eine Farbe bestreichen.
- Die Türme R bewegen sich geradlinig parallel zum Brettrand.
- Die Dame Q vereint die Zugarten von Turm und Läufer.
- Der König K kann nach allen Seiten nur einen Schritt ziehen. Aus der Grundstellung ist auch eine Rochade möglich.

Die Springer und Läufer werden auch als Leichtfiguren bezeichnet. Die Türme und Damen werden als Schwerfiguren bezeichnet, wobei die Dame doppelt gewichtet wird. Die Leichtfiguren und Schwerfiguren bilden zusammen die Klasse der Figuren, die Bauern stellen eine eigene Klasse dar.

1.1.3 Figurenbewertung

Die Grundeinheit für *Figurenwerte* und *Stellungswerte* ist der konstante Wert des Bauern.
Alle weiteren Figurenwerte und *Stellungswerte* werden in *1/100 Bauerneinheiten (centipawns cp)* angegeben. Der König bekommt keinen expliziten Materialwert, da er nie geschlagen werden kann. Aus Erfahrung haben sich bisher folgende Materialwerte und Korrekturen bei ungleicher Materialverteilung bewährt:

- 300 cp für die Leichfiguren Springer und Läufer.
- 500 cp für die Schwerfigur Turm
- 900 cp für die doppelt gewichtete Schwerfigur Dame
- Bei gleicher Anzahl Schwerfiguren und ungleicher Anzahl Leichtfiguren, wird der Seite mit der/den zusätzlichen Leichtfigure(n) ein Bonus von ~120 cp gegeben.
- Bei gleicher Anzahl Leichtfiguren und ungleicher Anzahl Schwerfiguren, wird der Seite mit der/den zusätzlichen Schwerfigure(n) ein Bonus von ~120 cp gegeben.
- Für 1 Schwerfigur gegen 2 Leichtfiguren wird ein Malus von ~120 cp für die Seite mit der zusätzlichen Schwerfigur vergeben.
- Bei 1 Schwerfigur gegen 1 Leichtfigur wird ein Bonus von ~60 cp für die Seite mit der zusätzlichen Schwerfigur vergeben, falls diese Seite eine Dame besitzt und der Gegner keine Dame besitzt.
- Für 2 Schwerfiguren gegen 3 Leichtfiguren wird ein Malus von ~120 cp für die Seite mit den zusätzlichen Schwerfiguren erwogen.
- Gleiche Anzahl Schwerfiguren und gleiche Anzahl Leichtfiguren, wobei eine Seite 1 Dame besitzt und die Gegenseite 2 Türme, so wird der Seite mit der Dame ein Bonus von ~120 cp gegeben.

1.1.4 Stellungsbewertung

Begründer der wissenschaftlichen Herangehensweise an das Schachspiel war der erste Weltmeister Wilhelm Steinitz. Eine der Hauptthesen seiner Theorie besagt, daß der Schachspieler einen Plan fassen sollte auf der Basis der objektiven Stellungsbewertung. Und diese Bewertung wiederum sollte basieren auf einer Reihe von Merkmalen, welche die Erfahrung vieler Tausend Schachpartien hervorgebracht hat. Einige dieser Merkmale wurden von Steinitz schon vor über einem Jahrhundert ausgewählt. Die meisten davon haben sich im Laufe der Zeit bewährt und wurden von anderen herausragenden Schachkoryphäen der Vergangenheit und Gegenwart modifiziert. Heute nennt man diese Merkmale die Elemente der Schachstrategie. Diese Elemente lauten:

- Das Zentrum
- Figurenentwicklung und Plazierung
- Bauernstruktur
- Starke und schwache Felder und Punkte
- Offene Linien und Diagonalen
- Vorteil des Läuferpaars
- Stellung des Königs

1.1.5 Bewertungsbeispiel

Nachfolgende Stellung vereint eine ausgeglichene Figurenbewertung mit einigen ausgeprägten Elementen der Stellungsbewertung:

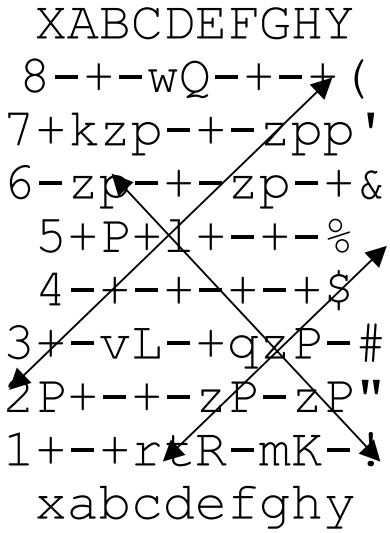


Abbildung 1-2 Schwarz in Vorteil trotz Materialgleichheit

Die Materialbilanz dieser Stellung ist nach den Kriterien der Figurenbewertung ausgeglichen

```
Materialbewertung = 0.
```

Ohne Stellungsbewertung wäre der deutliche Vorteil von Schwarz nicht zu erkennen. Während das Zentrum relativ ausgeglichen ist und sich auch die Bewertung der Bauernstruktur in der Waage hält, verfügt Schwarz über einen überwältigenden Königsangriff. Die an den Weissen König angrenzenden Felder **f3, g2, h3** sind schwache Felder, denen sich Schwarz bereits bemächtigt hat, um einen aussichtsreichen Königsangriff zu starten. Diese schwachen Felder können nicht von den Weissen Bauern verteidigt werden. Zudem kontrollieren die Schwarzen Figuren nahezu alle wichtigen Diagonalen. Weiss verfügt über keinerlei Gegenspiel, obwohl der Schwarze Königsschutz sehr schwach ist.

Je harmonischer die Elemente der Stellungsbewertung zueinander gewichtet sind, desto erfolgreicher kann eine geplante Strategie verwirklicht werden. Der englische Programmierer Mark Uniacke der Schachengine „HIARCS“ formulierte diese Problematik in einem Interview:

> *„Ein Bärenanteil der Zeit benötigt man für die ausgewogene Gewichtung der Bewertungseinheiten. Es ist sehr einfach spezifisches Wissen in Isoliertheit zu programmieren, aber sehr schwer zu bestimmen wie dieses neue Wissen dann in praktischen Partien mit den anderen Funktionen interagiert. Das macht die ganze Entwicklung sehr problematisch.“*

Eine vollständige Positionsbewertung setzt sich aus der Materialbewertung und den einzelnen gewichteten Elementen der Stellungsbewertung zusammen. In einigen Bewertungen finden zusätzlich Bewertungsasymmetrien Anwendung, um ausgeglichene Stellungen anzustreben, die dem Charakter einer Schachengine oder eines Schachspielers entgegenkommen.

So werden beispielsweise in Partien Mensch gegen Computer von den Computern offene und komplizierte Stellungen angestrebt, um ihre präzise Kombinationsfähigkeiten auszunutzen. Die Angabe des Positionswertes erfolgt aus Sicht der Seite am Zug, und kann aufgrund möglicher Asymmetrien nicht immer durch Vorzeichenwechsel auch für die Gegenseite gelten:

```
PositionswertAusSichtSeiteAmZug = Materialbewertung +
 StellungsspezifischerAsymmetrieWert +
(SkalierungErstesBewertungskriterium × WertErstesBewertungskriterium) +
(SkalierungZweitesBewertungskriterium × WertZweitesBewertungskriterium) +
....
(SkalierungLetztesBewertungskriterium × WertLetztesBewertungskriterium)
```

Auf eine Erläuterung der Bewertungskriterien wird hier verzichtet, eine gute Einführung findet man z.B. in Karpov, Mazukewitsch „Stellungsbeurteilung und Plan“, Nimzowitsch „Mein System“ oder Kmoch „Die Kunst der Bauernführung“.

1.1.6 Spielziel

Das Ziel des Spiels ist es, den gegnerischen König Matt zu setzen, d.h. so anzugreifen, dass er sich dem Geschlagenwerden nicht mehr entziehen kann. Wenn der am Zug befindliche Spieler keine seiner Figuren mehr ziehen kann und sein König nicht bedroht wird ist die Stellung Patt und das Spiel endet im Kräftegleichgewicht remis. Durch 3-fache Stellungswiederholung oder 50-faches Ziehen ohne irreversiblen Zug (Bauernzug oder Schlagzug) endet die Partie ebenfalls remis.

1.2 Grundlagen der Suche

1.2.1 Suchen in der Tiefe

Wenn ein Schachprogramm zu dem Schluss kommt, dass es mit der Dame einen Bauern schlagen sollte, muss es auch berechnen, ob der Gegner dafür die Dame oder eine andere Figur erobern wird. Die ursprüngliche Punktzahl für die Eroberung des Bauern

```
BauernWert = 100
```

kann nach Verlust der Dame

```
DamenWert = 900
```

auf

```
100 – 900 = -800
```

sinken, und so nach der Materialwertzählung zu einer ungünstigen Materialbilanz führen. Das Programm versucht also Züge zu machen, die seine Bewertungszahl maximieren. Der Gegner macht ebenfalls Züge, die seine Punktzahl maximieren. Das Programm versucht nun seinerseits, den Schaden, den der Gegner seiner Punktzahl zufügen kann, durch weitere Züge zu verringern. Dieser Prozess wird solange fortgesetzt, wie das Programm die geplante Serie von Zügen *(Variante)* durchleuchten kann.

Das Verfahren, durch Maximierung der Programmpunkte bei gleichzeitiger Minimierung der gegnerischen Antworten, Züge zu bestimmen, heisst *Minimax-Verfahren*.

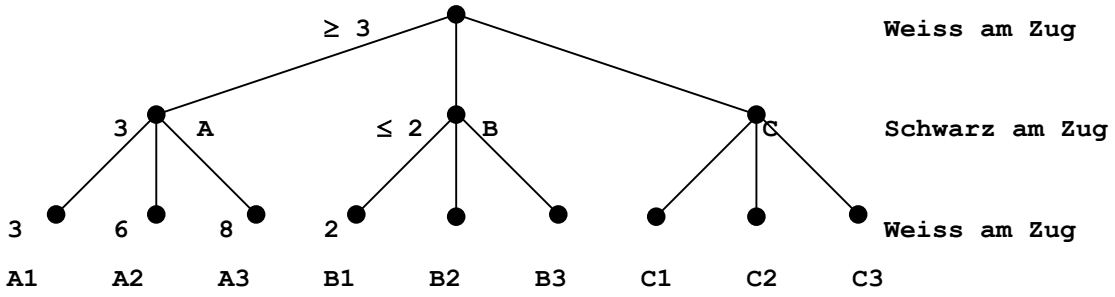


Abbildung 1-3 Suchbaum

In Abbildung 1-2 hat Weiss 3 Züge **(A, B, C)** zur Auswahl, Schwarz wiederum hat 3 Antwortzüge **(A1, A2, A3, B1, B2, B3, C1, C2, C3)** auf jeden Zug von Weiss. Da Weiss den Wert seiner Stellung maximiert und den Wert von Schwarz minimiert, weil Schwarz seinerseits maximiert, wird Zug **A** nach Durchrechnung seines Teilbaumes nach 2 Halbzügen mit 3 bewertet. Im diesem Suchbaum bewertet Weiss also Zug **A** nach dem *Minimax-Verfahren* mit

Minimum(A1, A2, A3) = Minimum(3, 6, 8) = -Maximum(-3, -6, -8) = 3

Da Weiss am Zug ist, und sich für den aus seiner Sicht besten Zug entscheiden wird (Maximierung), besitzt Weiss nach Berechnung von Zug **A** die Information, dass die aktuelle Stellung (Wurzelknoten) mindestens mit

Maximum(A = 3, B, C) = -Minimum(-A = -3, -B, -C) ≥ 3

zu bewerten ist.

1.2.2 Alpha-Beta und Nega-Max-Verfahren

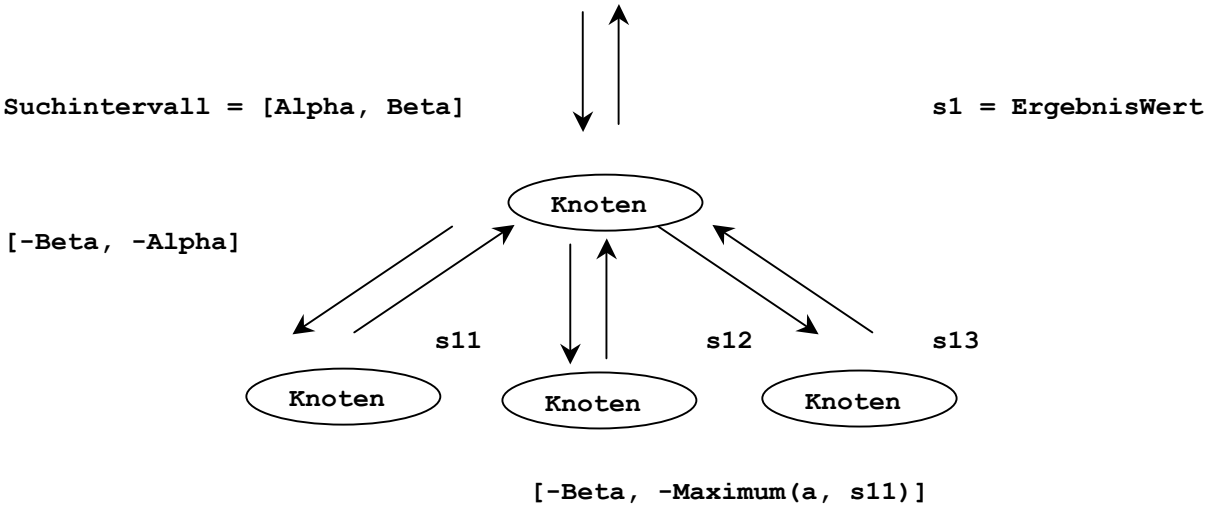


Abbildung 1-4 Datenfluss an einem Knoten nach *Ernst A. Heinz*

Die Bewertung des Wurzelknotens nach Durchrechnung von Zug **A** nach Abbildung 1-2 mit mindestens 3 bedeutet, dass Weiss die Berechnung der nächsten Züge mit einer Untergrenze durchführen kann. Diese Untergrenze 3 wird als *Alpha-Schranke* bezeichnet. Die Wurzelstellung besitzt folglich nach Berechnung von Zug **A** den Wert

WurzelWert ≥ Alpha = 3.

Nach weiterer Berechnung wird der Knoten **B1** des zweiten Teilbaumes mit 2 bewertet. Da in dieser Rechentiefe Schwarz am Zug ist, wird der Wert für Schwarz maximiert und für Weiss minimiert. Mit der in **Alpha** gespeicherten Information und der Bewertung von Zug **B1** mit

Wert = 2

besitzt nun auch Schwarz für diesen Unterbaum aus seiner Sicht eine Unterschranke mit

Alpha ≥ -2.

Das Suchfenster von Weiss

SuchFensterWeiss = [Alpha, Beta]

gilt auch für Schwarz durch Negation und Tauschen von Unterschranke und Oberschranke

SuchFensterSchwarz = [-Beta, -Alpha].

Der Datenfluss von der Wurzel ausgehend in den Knoten hinein und der Datenrückfluss vom Knoten zur Wurzel wird in Abbildung 1-3 dargestellt.

1.2.3 Beta-Pruning

Die in **Alpha** und **Beta** gespeicherten Bewertungen, also die Bewertungsunterschranke und die Bewertungsoberschranke an einem beliebigen Knoten werden eingesetzt, um Teilbäume vollständig wegzuschneiden, dieser Vorgang wird als *Pruning* bezeichnet. In Abbildung 1-2 hat Weiss nach der Berechnung von Zug **A** die Information, dass Zug **A** den Wert **3** besitzt. Zum Zeitpunkt der Berechnung ist Zug **A** auch der einzige und somit beste Zug, da Weiss über Zug **B** und Zug **C** noch keine Informationen besitzt. Diese Information wird als Bewertungsunterschranke von Weiss in **Alpha** gespeichert. Während der Bewertung von Knoten **B1** mit **2** und dem Datenrückfluss (nach Abbildung 1-3) zu dessen übergeordneten Knoten (Vaterknoten) **B**, kann Weiss die Bewertung von allen weiteren Kindknoten **B2** und **B3** vernachlässigen, da aus Sicht von Schwarz

Maximum(-B1 = -2, -B2, -B3) ≥ -2

nach oben aus dem **Alpha-Beta**-Suchfenster von Schwarz

SuchFensterSchwarz = [Alpha, Beta = -3]

mit

Maximum(-B1 = -2, -B2, -B3) ≥ -2 ≥ Beta = -3

herausfällt. Dieser Prozess des Beschneidens von Unterbäumen wird als *Beta-Pruning* bezeichnet und ist nur am Datenrückfluss vom Kindknoten zum Vaterknoten aktiv. Für alle Bewertungen eines Knotens (Wurzelknoten, Vaterknoten, Kindknoten) gilt

KnotenWert ∈ [-Matt, +Matt],

ausserdem gilt für die Schranken

Unterschranke = Alpha < Oberschranke = Beta
⇒ Unterschranke = Alpha ≤ Oberschranke - 1 = Beta – 1

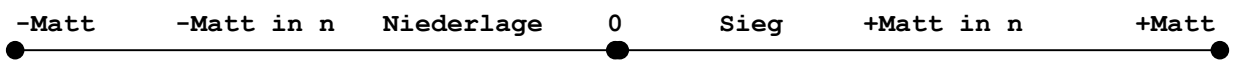
Daher gilt für die Bewertungsunterschranke

Unterschranke = Alpha ∈ [-Matt, Matt - 1]

und die Bewertungsoberschranke

Oberschranke = Beta ∈ [-Matt + 1, Matt],

woraus folgende Bewertungsachse resultiert:



Es ergeben sich drei Möglichkeiten für das Ergebnis eines Knotens:

- **KnotenWert ≤ Alpha → LowScore,** Bewertung liegt unterhalb des Suchfensters
- **KnotenWert ≥ Beta → HighScore,** Bewertung liegt oberhalb des Suchfensters
- **Alpha < KnotenWert < Beta → ExactScore,** Bewertung liegt im Suchfenster und liefert ein exaktes Ergebnis

1.2.4 Rechenbeispiel

Mit dem Minimax-Verfahren, dem Alpha-Beta-Suchfenster und dem Beta-Pruning ist jeder Rechenschritt eines konstruierten Minimax-Suchverfahrens nachzuvollziehen:

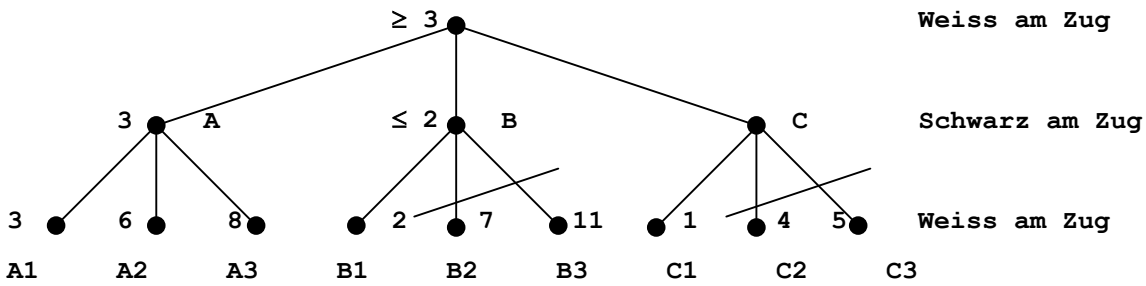


Abbildung 1-5 Suchbaum und Beta-Pruning

Da zu Beginn der Berechnung keine Informationen über den Wurzelknoten bekannt sind, beginnt Weiss die Berechnung von Zug **A** mit einem uneingeschränkten, grossen Alpha-Beta-Suchfenster über die komplette Bewertungsachse

SuchFensterWeiss = [Alpha = -Matt, Beta = Matt].

Dieses Suchfenster wird nach dem *Nega-Max-Verfahren* (Tauschen und Negieren der Alpha-Beta-Schranken nach Abbildung 1-3) an Schwarz übergeben

SuchFensterSchwarz = [Alpha = -Matt, Beta = Matt].

Nach Bewertung von Knoten **A1** aktualisiert Schwarz seine Unterschranke mit dem Wert von Knoten **A**

SuchFensterSchwarz = [Alpha = -3, Beta = Matt].

Die Knoten

A2 = -6 ≤ Alpha = -3

und

A3 = -8 ≤ Alpha = -3

sind für Schwarz schlechter und haben keinen Einfluss auf die Unterschranken und Oberschranken. Knoten **B** und dessen Unterbaum **B1, B2, B3** wird jetzt mit den Suchfenstern

SuchFensterWeiss = [Alpha = 3, Beta = Matt]

aus Sicht von Weiss, und

SuchFensterSchwarz = [Alpha = -Matt, Beta = -3]

bewertet.
Nach der Bewertung von Knoten **B1 = 2** wird die Berechnung von Knoten **B2** und **B3** wegen

B1 = -2 ≥ Beta = -3

nach dem Beta-Pruning-Mechanismus abgebrochen. Die Bewertung von Knoten **C** erfolgt analog mit

C1 = -1 ≥ Beta = -3

und wird ebenfalls nach dem Beta-Pruning-Mechanismus abgebrochen.

2 Bitboards

2.1 Grundlagen

2.1.1 64-Bit-Zahlen

Die heute üblichen Mikroprozessoren sind sogenannte 32-Bit-Prozessoren. Der grundlegende Datentyp solcher Prozessoren besteht aus 32 Bit. Die meisten 32-Bit-Rechenoperationen werden daher in einem einzigen Taktzyklus erledigt. Da *Bitboards* jedoch 64-Bit-Zahlen sind, werden bei der Programmierung keine echten 64-Bit-Zahlen verwendet, sondern nur eine Simulation solcher Zahlen, durch Zusammenfassung zweier aufeinanderfolgender 32-Bit-Zahlen. Da das Schachbrett aus 64 Feldern besteht, kann genau dieser Datentyp sehr effizient zur Darstellung des Schachbrettes verwendet werden. Jedes Bit dieses 64-Bit-Zahl steht für ein Feld des Schachbrettes. Da aber ein Bit lediglich die Information über Vorhandensein **`(Wert = 1)`** oder Nichtvorhandensein **`(Wert = 0)`** einer Figur liefert, lässt sich auf diese Weise nicht Feststellen welche Figur auf einem Feld des Schachbrettes sitzt. Zur Lösung dieses Problems werden mehrere Bitboards verwendet, für jeden Figurentyp eines:

- **`Bauern`**
- **`Springer`**
- **`Läufer`**
- **`Türme`**
- **`Damen`**
- **`Könige`**
- **`Weisse Figuren`**
- **`Schwarze Figuren`**

2.1.2 Logikoperationen

Gewöhnlich repräsentiert das erste Bit auch das erste Feld des Schachbrettes **`a1`**, das zweite Bit **`a2`** bis zum letzten Bit, dem Feld **`h8`**. Das Bauern-Bitboard in der Grundstellung sieht wie folgt aus:

```
(Bauern)
00000000  (a8....h8)
11111111   .      .
00000000   .      .
00000000   .      .
00000000  (a4....h4)
00000000   .      .
11111111   .      .
00000000  (a1....h1)
```

Zum Rechnen mit Bitboards werden die unären und binären bitweisen Logikoperationen benötigt.

- Bitweises Linksshiften **LEFTSHIFT**
- Bitweises Rechtsshiften **RIGHTSHIFT**
- Einer-Komplement **COMPLEMENT**
- Bitweises Und **AND**

AND	0	1
0	0	0
1	0	1

Abbildung 2-1 Funktionstabelle AND

- Bitweises Oder **OR**

OR	0	1
0	0	1
1	1	1

Abbildung 2-2 Funktionstabelle OR

Der Vorteil der Bitboards, also dem Darstellen des gesamten Brettes in einer Zahl, besteht in der nahezu unbegrenzten Möglichkeit der Datengewinnung durch Zahlenmanipulation. Zum Beispiel lässt sich durch Addition von Bauern-Bitboard und Turm-Bitboard ein Bauern-Turm-Bitboard erzeugen:

```
(Bauern)    (oder)     (Türme)          (Bauern oder Türme)
00000000               10000001         10000001
11111111               00000000         11111111
00000000               00000000         00000000
00000000               00000000         00000000
00000000    OR         00000000    =    00000000
00000000               00000000         00000000
11111111               00000000         11111111
00000000               10000001         10000001
```

Durch Verwendung weiterer Logikoperationen lassen sich komplexere Informationen, wie zum Beispiel sämtliche legale Bauernzüge von Weiss in nur wenigen Schritten berechnen.
So lassen sich die Weissen Bauern herausfiltern, durch bitweise **AND**-Verknüpfung von Bauern-Bitboard und Weisse-Figuren-Bitboard:

```
(Bauern)    (und)      (Weisse Figuren) (Weisse Bauern)
00000000               00000000         00000000
11111111               00000000         00000000
00000000               00000000         00000000
00000000               00000000         00000000
00000000    AND        00000000    =     00000000
00000000               00000000         00000000
11111111               11111111         11111111
00000000               11111111         00000000
```

Das berechnete Weisse-Bauern-Bitboard 8 Felder nach links schieben mittels Linksshift, um sämtliche Weisse Bauern ein Feld vorzurücken:

```
(Weisse Bauern) (Linksshift um 8 Bitstellen) (Weisse Bauernzüge)
00000000                                     00000000
00000000                                     00000000
00000000                                     00000000
00000000                                     00000000
00000000    LEFTSHIFT 8   =                  00000000
00000000                                     11111111
11111111                                     00000000
00000000                                     00000000
```

Bauernzüge sind nur legal, wenn auf dem Zielfeld keine Figur steht. Diese Logikoperationen lassen sich nochmals kompakt darstellen:

```
WeisseBauern = Bauern AND FigurenWeiss
WeisseBauernZüge = BauernWeiss LEFTSHIFT 8
UnfreieFelder = FigurenWeiss OR FigurenSchwarz
FreieFelder = COMPLEMENT(UnfreieFelder)
LegaleWeisseBauernzüge = BauernZügeWeiss AND FreieFelder
```

oder zu einer einer Rechenoperation zusammengefasst:

```
LegaleBauernzügeWeiss = ((Bauern AND FigurenWeiss) LEFTSHIFT 8) AND
                        COMPLEMENT(FigurenWeiss OR FigurenSchwarz)
```

Analog zu diesem Beispiel lassen sich beliebig komplexe Informationen sehr schnell berechnen, sofern die Gleiteigenschaften von Läufer, Turm und Dame vernachlässigt werden.

2.2 Bitboard-Rotationen

2.2.1 Look-Up-Tables

Um mit den Gleiteigenschaften von Läufer, Turm und Dame genauso effizient rechnen zu können wie mit den übrigen Figuren reichen die bisherigen acht Bitboards nicht aus.

Da sich zum Beispiel die Turmzüge aus horizontalen und vertikalen Zügen zusammensetzen, werden diese auch separat berechnet.

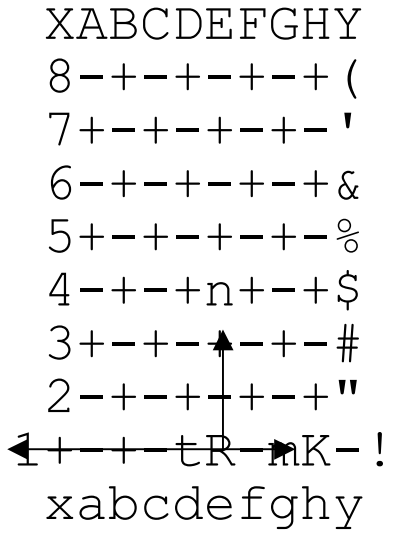


Abbildung 2-3 Gleiteigenschaften des Turmes

In einer Tabelle werden für alle 64 Felder des Brettes die horizontalen Züge des Turmes eingetragen. Um die Gleiteigenschaften zu berücksichtigen, werden für jeden dieser 64 Tabelleneinträge weitere 256 Untereinträge berechnet, da eine Reihe aus 8 Feldern besteht, und somit 2^8=256 Figurenkombinationen auf jeder Reihe möglich sind.

Für die Figurenverteilungen auf der jeweiligen Reihe wird lediglich die Information über Vorhandensein oder Nichtvorhandensein einer Figur benötigt. Um die horizontalen Gleitzüge aus dieser einmal berechneten Tabelle *(Look-Up-Table)* auszulesen wird der Feldindex und die Figurenverteilung auf der entsprechenden Reihe benötigt:

```
0 ≤ Index1 = FeldIndex ≤ 63
0 ≤ Index2 = HorizontalFigurenverteilung(FeldIndex) ≤ 255
HorizontaleZüge = HorizontalTabelle[Index1][Index2]
```

Die vertikalen Turmzüge werden analog zu den Horizontalen berechnet. Es wird nur zusätzlich ein um 90 Grad nach links oder rechts gedrehtes Bitboard benötigt, da es nicht möglich ist die Figurenverteilung aus einer Vertikalen oder einer Diagonalen zu berechnen. Es wird also für jede weitere Gleitrichtung ein explizites Bitboard benötigt, welches die Bitinformation über Vorhandensein oder Nichtvorhandensein in einer ununterbrochenen Reihe darstellt.

```
0 ≤ Index1 = FeldIndex ≤ 63
0 ≤ Index2 = VertikaleFigurenverteilung(FeldIndex) ≤ 255
VertikaleZüge = VertikaleTabelle[Index1][Index2]
```

Das Bitboard für alle Turmzüge setzt sich nun aus den Horizontalzügen und den Vertikalzügen zusammen:

```
Turmzüge = HorizontalZüge OR VertikalZüge
```

Diese Züge sind natürlich nur legal, wenn der Turm auf ein Feld zieht, welches nicht von eigenen Figuren besetzt ist:

```
LegaleTurmzügeWeiss = (HorizontalZüge OR VertikalZüge) AND
                      COMPLEMENT(FigurenWeiss)
```

2.2.2 45 Grad-Rotationen

Naheliegend ist die Drehung des Schachbrettes um 45 Grad nach links und um 45 Grad nach rechts, um das diagonale Gleiten des Läufers mit Hilfe der *Look-Up-Tables* in zwei Rechenschritten zusammenzusetzen.

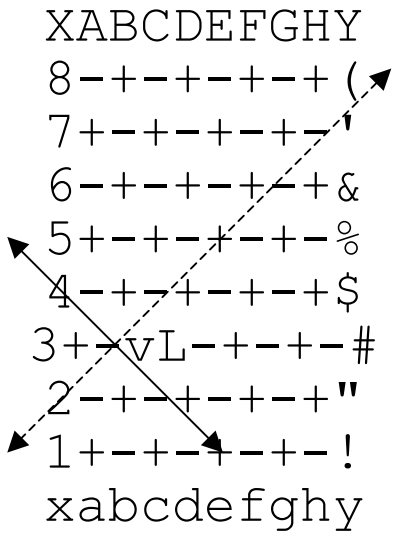


Abbildung 2-4 Gleiteigenschaften des Läufers

Das um 45 Grad nach rechts gedrehte Bitboard stellt die unterschiedlich langen Diagonalen (gestrichelte Linie) horizontal dar. Hieraus ergibt sich eine bijektive Abbildung, welche die Diagonalen als Reihen abbildet. So wird zum Beispiel die gestrichelte Diagonale **a1-h8** in Abbildung 2-4 als erste Reihe abgebildet. Für die Abbildungsvorschrift ***f*** für die 45 Grad-Rechtdrehung ***f: Brett → 45-Grad-Rotiertes-Brett*** ergibt sich:

```
 0,  9, 18, 27, 36, 45, 54, 63,
     8, 17, 26, 35, 44, 53, 62,  7,
        16, 25, 34, 43, 52, 61,  6, 15,
            24, 33, 42, 51, 60,  5, 14, 23,
                32, 41, 50, 59,  4, 13, 22, 31,
                    40, 49, 58,  3, 12, 21, 30, 39,
                        48, 57,  2, 11, 20, 29, 38, 47,
                            56,  1, 10, 19, 28, 37, 46, 55
```

Einzig erschwerend kommt die unterschiedliche Länge der Diagonalen hinzu, welche aber bereits bei der Berechnung der Look-Up-Tables berücksichtigt wird. Das diagonale Gleiten lässt sich ebenso effizient berechnen, wie das horizontale und das vertikale Gleiten:

```
0 ≤ Index1 = FeldIndex ≤ 63
0 ≤ Index2 = RechteDiagonaleFigurenverteilung(FeldIndex) ≤ 255
RechteDiagonaleZüge = RechteDiagonaleTabelle[Index1][Index2]
```

Das Bitboard für alle Läuferzüge setzt sich nun aus den beiden Diagonalzügen zusammen:

```
Läuferzüge = RechteDiagonaleZüge OR LinkeDiagonaleZüge
```

2.2.3 Rotierte Bitboards

Die ursprünglichen 8 Bitboards müssen jetzt noch um das 90-Grad gedrehte, und die beiden um jeweils 45-Grad gedrehten Bitboards erweitert werden. Das Bitboard für die horizontalen Gleitbewegungen muss nicht explizit berechnet werden, da es sich aus den Figuren-Bitboards von Weiss und Schwarz berechnen lässt:

- **Bauern**
- **Springer**
- **Läufer**
- **Türme**
- **Damen**
- **Könige**
- **Weisse Figuren**
- **Schwarze Figuren**
- **Figuren rotiert um 45° nach links**
- **Figuren rotiert um 45° nach rechts**
- **Figuren rotiert um 90° nach links oder rechts**
- **Figuren rotiert um 0° folgt aus (WeisseFiguren OR SchwarzeFiguren)**

3 Hashverfahren

3.1 Hashschlüssel

3.1.1 Allgemeines

Die Verwaltung grosser Datenmengen und das schnelle Wiederfinden von Informationen ist seit jeher ein zentrales Problem der Informatik. Daher stellt sich die Frage, ob es nicht Verfahren gibt, bei denen der Suchaufwand völlig unabhängig ist von der Grösse des Datenvolumens, und die gibt es tatsächlich, es sind dies die *Hashverfahren (Rolf Isernhagen)*.

3.1.2 Problemstellung

Im Verlauf der Suche wird ein Schachprogramm mehrere Millionen Stellungen untersuchen, wobei in jeder Suche viele Stellungen mehrfach untersucht werden müssen. Das Hashverfahren für Schachprogramme soll die zuletzt untersuchten Stellungen in einem Hashspeicher ablegen, und bei Bedarf auf diese Daten zurückgreifen, um den Suchvorgang zu beschleunigen.

Die Stellung nach Abbildung 3-1 geht ursprünglich aus der Französischen Eröffnung hervor und entsteht durch die Zugfolge:

```
1. e2-e4  e7-e6
2. d2-d4
```

```
XABCDEFGHY
8rsnlwqkvlntr(
7zppzpp+pzpp'
6-+-+p+-+&
5+-+-+-+-%
4-+-zPP+-+$
3+-+-+-+-#
2PzPP+-zPPzP"
1tRNvLQmKLsNR!
xabcdefghy
```

Abbildung 3-1 Französische Eröffnung

Allerdings kann diese Stellung auch durch andere Zugfolgen Zustande gekommen sein:

```
1. d2-d4  e7-e6
2. e2-e4
```

Hieraus ist leicht zu erkennen, dass eine Stellung in einer Hashtabelle unabhängig von der Zugfolge, durch welche sie entstanden ist, gespeichert werden muss, um effizient wiederbenutzt werden zu können.

3.1.3 Berechnung eines Hashschlüssels

Um die komplette Stellung zu kodieren, wird für jeden Figurentyp und jedes Feld, also 12 Figuren und 64 Felder, eine hinreichend grosse Zufallszahl benötigt. Für diese **12 × 64 = 768** Zufallszahlen ist ein 64-Bit-Ganzzahldatentyp am geeignetesten. Zur Berechnung eines Schlüssels müssen diese Feld -Figurentyp-Zufallszahlen miteinander mit dem **Exklusiven-Oder (XOR-Bitoperation)** verknüpft werden. Da diese Bitoperation kommutativ ist, spielt die Reihenfolge bei der Berechnung des Hashschlüssels keine Rolle.

XOR	0	1
0	0	1
1	1	0

Abbildung 3-2 Funktionstabelle XOR

Durch zweifache **XOR**-Operation mit derselben Zufallszahl erhält man wieder den Ausgangsschlüssel, da jede Zufallszahl zu sich selbst durch diese Operation Inverse ist. Diese Eigenschaft der **XOR**-Funktion ermöglicht inkrementelles Zugausführen und Zugzurücknehmen ähnlich dem Zugausführen und Zugzurücknehmen auf dem Schachbrett, durch die entsprechenden **XOR**-Operationen mit den Feld-Figurentyp-Zufallszahlen.

3.1.4 Rechenbeispiel

Die Stellung in Abbildung 3-3 setzt sich aus einem Weissen König auf **e2** und einem Schwarzen König auf **f5** zusammen. Durch Verknüpfung der diesen Figuren zugehörigen Feld-Figurentyp-Zufallszahlen lässt sich der Hashschlüssel für diese Stellung berechnen:

```
Hashschlüssel = FeldFigurZufallszahl[e2][KönigWeiss] XOR
                FeldFigurZufallszahl[f5][KönigSchwarz]
```

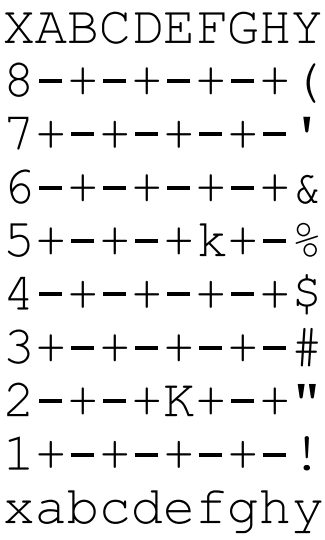


Abbildung 3-3 Beispielstellung für Hashschlüsselberechnung

Ändert sich die Stellung durch Ausführen eines Zuges, so kann der Hashschlüssel wegen der Inverseneigenschaft der **XOR**-Funktion sehr effizient inkrementell berechnet werden. Zieht beispielsweise der Weisse König in Abbildung 3-3 von Feld **e2** nach Feld **e3**, so muss der Weisse König von Feld **e2** entfernt werden und auf das Feld **e3** gesetzt werden. Da der Hashschlüssel vor Ausführen des Zuges **e2-e3** definiert ist als,

```
HashschlüsselVorZug = FeldFigurZufallszahl[e2][KönigWeiss] XOR
                      FeldFigurZufallszahl[f5][KönigSchwarz]
```

so wird der Weisse König von Feld **e2** inkrementell durch die Inverseneigenschaft entfernt

```
HashschlüsselNachEnfernenVonKönigE2 = HashschlüsselVorZug XOR
                      FeldFigurZufallszahl[e2][KönigWeiss]
```

und anschliessend auf das Feld **e3** gesetzt

```
HashschlüsselNachZug = HashschlüsselNachEnfernenVonKönigE2 XOR
                       FeldFigurZufallszahl[e3][KönigWeiss].
```

Zusammengefasst und verallgemeinert ergeben sich zwei Bitoperationen zur inkrementellen Berechnung eines Hashschlüssels bei Zugausführungen:

```
HashschlüsselNachZug = HashschlüsselVorZug XOR
                       FeldFigurZufallszahl[FeldVon][Figurentyp] XOR
                       FeldFigurZufallszahl[FeldNach][Figurentyp]
```

Die Verschlüsselung der Bauernumwandlung und der Rochade erfolgen analog. Einzig der Enpassant-Status und die Rochaderechte müssen explizit in den Hashschlüssel hineinkodiert werden.

3.2 Hashspeicher

3.2.1 Hasheinträge

Zusätzlich zum Hashschlüssel, welcher den Hasheintrag (fast) eindeutig identifiziert, müssen noch weitere Daten der Suche gespeichert werden, um in der weiteren Suche effizient und fehlerfrei wiederverwendet werden zu können. Diese Daten beschränken sich weitgehend auf den Kontext der Suche zum Zeitpunkt der Speicherung:

- Welcher Knoten wurde am genauesten berechnet → der Beste Zug
- Wie tief war die Suche → Distanz der Suche
- Untergrenze und Obergrenze des Suchfensters → **`HighScore, LowScore`** oder **`ExactScore`**
- Stellungsbewertung nach Durchsuchen des Unterbaumes → **`KnotenWert`**

3.2.2 Speicheradressen

Aus dem berechneten Hashschlüssel für eine Stellung kann jetzt die Speicheradresse berechnet werden. Die Bestimmung der Speicheradresse ist eine Abbildung von dem Hashschlüssel in die Anzahl der zur Verfügung stehenden Speicheradressen, die abhängig ist von der gewählten Speichergrösse und der spezifischen Grösse für einen Hasheintrag. Für diese Abbildung stehen zwei Möglichkeiten zur Verfügung:

- Ausblenden von Bitstellen des Hashschlüssels mittels des logischen *Und*-Operators
- Divisionsrest-Verfahren mit dem *Modulo*-Operator

Ausserdem sind die Anforderungen an eine Hashfunktion

- Die berechneten Adressen sollen gleichmässig über den Adressbereich gestreut werden
- Der zeitliche Berechnungsaufwand sollte angemessen sein

Das *Ausblenden der Bitstellen* ist zwar recheneffizienter, allerdings lassen sich nur Adressräume in der Grösse von Zweierpotenzen allokieren. Stehen beispielsweise **`1 Megabyte`** Speicher zur Verfügung und beträgt die Grösse eines Hasheintrages **`16 Byte,`** so kann der vorhandene Speicher wegen

$$\texttt{VorhandenerSpeicher} = \texttt{GrösseSpeicherMB} \times \texttt{BytesProMB}$$
$$\texttt{GenutzterSpeicher} = \texttt{GrösseProHasheintrag} \times \texttt{GrösseAdressraum}$$
$$= \texttt{GrösseProHasheintrag} \times 2^{\texttt{x}}$$
$$\rightarrow \texttt{x} = \texttt{log(VorhandenerSpeicher / GenutzterSpeicher) / log(2)}$$
$$\rightarrow \texttt{GenutzterSpeicher} \leq \texttt{VorhandenerSpeicher}$$

vollständig genutzt werden, weil

VorhandenerSpeicher = 1 × (1024 × 1024) = 1048576 Bytes

und sich nach obiger Formel für **x** ergibt

x = 16

und folglich der genutzte Speicher so gross ist wie der vorhandene Speicher

GenutzterSpeicher = 16 × 65536 = 1048576 Bytes
→ GenutzterSpeicher = VorhandenerSpeicher.

Bei **1.5 Megabyte** vorhandenem Speicher könnte nach obiger Rechnung ebenfalls nur **1 Megabyte** genutzt werden. Für das *Divisionsrestverfahren* existiert dieses Problem nicht und es gilt immer für jede beliebige vorhandene Speichergrösse

GenutzterSpeicher = VorhandenerSpeicher.

3.3 Speicherorganisation

3.3.1 Steigerung der Effizienz

Aufgrund der eingeschränkten Speicherressourcen und des enormen Speicherbedarfs während der Variantensuche, kommt es sehr bald zu Adresskollisionen. Als Folge dieser Adresskollisionen müssen bereits bestehende Tabelleneinträge überschrieben werden, welche in nachfolgenden Berechnungen nicht mehr zur Verfügung stehen und möglicherweise neu berechnet werden müssen.

Eine signifikante Effizienzsteigerung wird durch die Vergabe eines Gütekriteriums für jeden Hashtabelleneintrag zum Zeitpunkt seiner Speicherung in der Tabelle erreicht. Da jeder Tabelleneintrag Informationen über eine Stellung beinhaltet, ist es naheliegend das Gütekriterium von dem benötigten Rechenaufwand für eine Stellung abhängig zu machen. Das geeigneteste Gütekriterium hierfür ist die *Rechentiefe*, mit der eine Stellung berechnet wurde.

3.3.2 Gütekriterien

Eine effiziente Speicherstrategie lässt sich sehr gut an einem Modellbeispiel herleiten. Vereinfacht wird nachfolgend mit einem 32-Bit-Schlüssel abgebildet auf einen 16-Bit-Adressraum gerechnet.
Für einen 16-Bit-Adressraum und einer spezifischen Grösse von **`16 Byte`** pro Hasheintrag ergibt sich ein Speicherbedarf von

```
GenutzterSpeicher = GrösseProHasheintrag × GrösseAdressraum
                  = 16 × 65536 = 1048576 Bytes = 1 Megabyte
```

Die zu speichernde Stellung wird nachfolgend mit **`A`** bezeichnet. Der inkrementell berechnete Schlüssel von Stellung **`A`** sei

```
HashschlüsselStellungA = 1.234.567.890
```

hieraus ist die Speicheradresse durch *Ausblenden der Bitstellen* mittels Speichermaske zu berechnen

```
Speichermaske = (GrösseAdressraum – 1)
SpeicheradresseStellungA = HashschlüsselStellungA AND Speichermaske

Speichermaske = (65536 – 1) = 65535.
```

Aus der Darstellung von Speichermaske, Hashschlüssel und Speicheradresse in Binardarstellung ergibt sich

```
Speichermaske = 11111111.11111111
HashschlüsselStellungA = 1001001.10010110.00000010.11010010
SpeicheradresseStellungA = 1001001.10010110.00000010.11010010 AND
                                          11111111.11111111
                         = 0000000.00000000.00000010.11010010.
```

Die Speicheradresse für Stellung **A** umgewandelt in das Dezimalsystem ergibt wieder

```
SpeicheradresseStellungA = 722.
```

Speicheradresse	721	722	723	724
Hashschlüssel	3.446.781.097	1.771.438.802	2.039.874.258	1.234.830.034
Rechentiefe	4	7	2	5
weitere Daten...				

Abbildung 3-4 Speicherausschnitt

Angenommen Stellung **A** wurde mit einer Rechentiefe von 3 Halbzügen berechnet, so hat diese Stellung nach dem Gütekriterium *Rechentiefe* eine niedrigere Güte, als die bereits in Adresse **722** eingetragenen Stellung mit ihren Daten wegen

```
RechentiefeStellungA < Rechentiefe[StellungInAdresse722].
```

Ein guter Kompromiss aus Recheneffizienz und Speichereffizienz ist eine einfache lineare Sondierung. Zum Beispiel ist es praktisch in den nächsten Adressen **723** und **724** usw. einen Eintrag mit geringerer Güte *(Rechentiefe)* zu suchen. Es folgen also weitere Vergleichoperationen

```
if (Rechentiefe[StellungInAdresse723] < RechentiefeStellungA) {
  ÜberschreibeStellungInHashtabelle();
}
```

Kann keine Speicheradresse gefunden werden, welche obigen Kriterien genügt, so ist keine effiziente Speicherung möglich. Hieraus folgt der Basisalgorithmus für die Speicherorganisation mit Gütekriterium *Rechentiefe* in C-Code:

```
void Speicherorganisation(int Tiefe) {
  Speicheradresse = Hashschlüssel & Speichermaske;
  if (Hashschlüssel[Speicheradresse] == Hashschlüssel) {
    /* Stellung ist bereits eingetragen und wird aktualisiert */
    AktualisiereStellungInHashtabelle();
  }
  else if (Rechentiefe[Speicheradresse] < Tiefe) {
    /* Stellung überschreibt älteren Eintrag mit niedrigerer Tiefe */
    ÜberschreibeStellungInHashtabelle();
  }
  /* Zur nächsten Speicherstelle wechseln und Prozedur wiederholen */
  else { ... }
}
```

4 Variantenberechnung

4.1 Das Minimax-Verfahren

4.1.1 Zeitkomplexität

Ein wichtiges Beurteilungskriterium für die Bewertung von Rechenalgorithmen ist das Zeitverhalten. Es ist unmöglich, einen konkreten Wert für die Zeit anzugeben, die für eine Berechnung benötigt wird. Die benötigte Zeit hängt zumindest von den aktuellen Parametern und von dem Prozessor ab, auf dem der entsprechende Code abläuft. Aber es lässt sich eine qualitative Aussage bezüglich des Zeitverhaltens eines Algorithmus machen *(Rolf Isernhagen).*

Für ein proportionales Zeitverhalten einer einfachen Summationsfunktion

summe(n) = 1 + 2 + 3 + ... + (n - 1) + n

gilt zum Beispiel

$t_{summe} \sim n$ **für** $n \to \infty$

oder als allgemeine Darstellung dieses Zusammenhangs wird auch folgende Notation verwendet

$T_{summe} = O(n)$.

Bei der *Variantenberechnung* ist der Zeitaufwand abhängig von dem Verzweigungsfaktor des Suchbaumes, der von der Beschaffenheit des Rechenalgorithmus und der zu lösenden Problemstellung abhängt, und von der gewählten Rechentiefe

$t_{variantenberechnung} \sim Verzweigungsfaktor^{Tiefe}$

$T_{variantenberechnung} = O(Verzweigungsfaktor^{Tiefe})$.

Die Minimierung des Verzweigungsfaktors bei gleichbleibender Rechengenauigkeit ist daher oberstes Ziel in der Variantenberechnung.

4.1.2 Der Horizonteffekt

Unabhängig vom Verzweigungsfaktor wird jede Suche bei einer bestimmten Rechentiefe enden. Die hieraus berechneten Varianten sind daher meist fehlerhaft, da Ereignisse oberhalb dieser Rechentiefe nicht in der Variantenberechnung berücksichtigt werden. Diese Ereignisse liegen oberhalb des *Rechenhorizontes* und sind daher eine der Hauptfehlerquellen bei der Variantenberechnung.

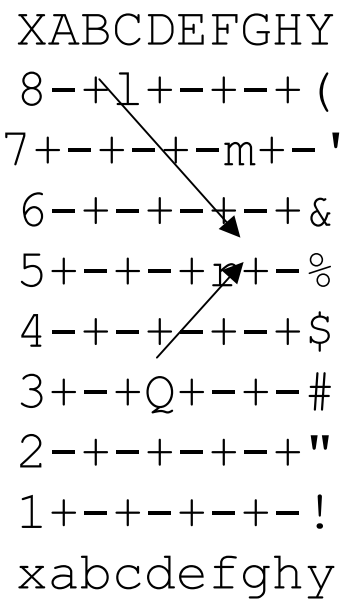


Abbildung 4-1 Horizonteffekt bei Tiefe 1

Bei einer Suchtiefe von 1 würde Weiss in obiger Abbildung mit der Dame auf **d3** den schwarzen Turm auf **f5** schlagen, statt einen anderen Zug in Erwägung zu ziehen, wegen

```
Suchwert(d3d5) = 0
...
Suchwert(d3d8) = 0
...
Suchwert(d3f5) = Wert(Turm) > 0
```

```
Maximum(Suchwert(d3d5), ..., Suchwert(d3d8), ..., Suchwert(d3f5))
                = Suchwert(d3f5) = Wert(Turm) > 0.
```

Erst ab Rechentiefe 2 kann durch das rekursive Minimax-Verfahren erkannt werden, dass nach Schlagen des schwarzen Turmes die weisse Dame verloren geht, wegen

```
Suchwert(d3d5) = -Maximum(Suchwert(alleFolgezügeSchwarz))
               = 0 – Wert(Dame) < 0
...
Suchwert(d3d8) = -Maximum(Suchwert(alleFolgezügeSchwarz))
               = 0 – 0
...
Suchwert(d3f5) = -Maximum(Suchwert(alleFolgezügeSchwarz))
               = Wert(Turm) – Wert(Dame) < 0
```

```
Maximum(Suchwert(d3d5), ..., Suchwert(d3d8), ..., Suchwert(d3f5))
               = Suchwert(d3d8) = 0.
```

4.1.3 Die Ruhesuche

Bereits von *Claude Shannon* wurde gefordert, nur *ruhige Stellungen* zu bewerten. Eine Stellung kommt erst zur Ruhe, wenn es keine materialbilanzverändernde Züge mehr gibt. Hierfür müssen lediglich die Schlagzüge (und die Umwandlungen) berücksichtigt werden.

Schliesst man an die konventionelle Minimax-Suche eine *Ruhesuche* an, so wird bereits bei Rechentiefe 1 der falsche Zug in Abbildung 4-1 **`d3f5`** vermieden. Eine klassische Umsetzung dieses Verfahrens demonstriert nachfolgender Beispielalgorithmus:

```
int Ruhesuche(int Farbe, int Tiefe, int Alpha, int Beta) {
  /* Weiss am Zug */
  if (Farbe == WEISS) {
    int Wert, Zug;
    /* Position aus Sicht von „Farbe" bewerten */
    Wert = Positionsbewertung(Farbe);
    if (Wert >= Beta) return Beta;
    if (Wert > Alpha) Alpha = Wert;
    /* Zugschleife durchlaufen */
    while ((Zug = NächsterSchlagzug())) {
      /* Zug ausführen, Maximum der Stellung berechnen, Zug zurücknehmen */
      ZugAusführen(Zug);
      Alpha = Maxium(Alpha, -Ruhesuche(SCHWARZ, Tiefe + 1, -Beta, -Alpha));
      ZugZurücknehmen(Zug);
      /* Ruhesuche abbrechen, wenn Unterschranke >= Oberschranke (Alpha-Beta-Pruning) */
      if (Alpha >= Beta) return Beta;
    }
    return Alpha;
  }
  /* Schwarz am Zug (analog „zu Weiss am Zug", nur mit getauschten Farben rechnen) */
  else { ... }
}
```

Dieser ebenfalls rekursive Prozess wird solange vertieft, bis entweder die Positionsbewertung die Oberschranke übertrifft

```
if (Wert >= Beta) return Beta;
```

oder keine Schlagzüge mehr existieren, und die Ruhesuche zu einer *ruhigen Stellung (quiescent position)* geführt hat.

4.1.4 Die Baumsuche

Der Minimax-Prozess mit Obergrenze und Untergrenze, sowie der rekursiven Vertiefung und dem Übergang in die Ruhesuche lässt sich ebenfalls als Beispielalgorithmus entwickeln:

```
int Baumsuche(int Farbe, int Tiefe, int Distanz, int Alpha, int Beta) {
  /* Weiss am Zug */
  if (Farbe == WEISS) {
    int Zug;
    /* Zugschleife durchlaufen */
    while ((Zug = NächsterZug())) {
      /* Zug ausführen, Variante berechnen, Zug zurücknehmen */
      ZugAusführen(Zug);
      /* Vor dem Suchhorizont -> Baumsuche */
      if (Distanz - 1 > 0) {
        Alpha = Maximum(Alpha, -Baumsuche(SCHWARZ, Tiefe + 1, Distanz - 1, -Beta, -Alpha));
      }
      /* Suchhorizont erreicht -> Ruhesuche */
      else {
        Alpha = Maximum(Alpha, -Ruhesuche(SCHWARZ, Tiefe + 1, -Beta, -Alpha));
      }
      ZugZurücknehmen(Zug);
      /* Baumsuche abbrechen, wenn Unterschranke >= Oberschranke (Alpha-Beta-Pruning) */
      if (Alpha >= Beta) return Beta;
    }
    return Alpha;
  }
  /* Schwarz am Zug (analog zu „Weiss am Zug", nur mit getauschten Farben rechnen) */
  else { ... }
}
```

Während der Vertiefung nimmt die Tiefe zu und die Distanz ab. Nach dem Nega-Max-Verfahren werden die Untergrenze und die Obergrenze in jeder weiteren Vertiefung getauscht und negiert.
In jeder Rechentiefe werden die möglichen Züge ausgeführt, rekursiv vertieft und zurückgenommen.
Die Errechnung des Maximums und der Abbruch der Rekursion durch den Pruning-Prozess sind analog zur Ruhesuche.

4.2 Heuristische Verfahren

Trotz den deterministischen Pruning-Verfahren im Minimax-Prozess und den Hashverfahren bleibt der Verzweigungsfaktor sehr gross. Für eine leistungsfähige Variantenberechnung müssen zusätzlich *heuristische Verfahren* eingesetzt werden.

Gezielte Abschneidungen von Unterbäumen werden wegen ihrer Eigenschaft, Unterbäume bereits vor dem Suchprozess wegzuschneiden, auch als *Vorwärtspruning* bezeichnet. *Erweiterungen* von Unterbäumen ermöglichen es ausserdem, heuristisch interessante Varianten tiefer zu untersuchen und den Horizont interessanter Unterbäume zu verschieben. Diese heuristischen Verfahren zusammengenommen ermöglichen allein durch Abschätzungen eine deutlich tiefere und somit genauere Variantensuche.

4.2.1 Nullmoves

Dieses Vorwärtspruning-Verfahren beruht auf der einfachen Idee, dass es in jeder Stellung mindestens einen Zug gibt, der besser ist als das *Nichtziehen (nullmoving).* Diese heuristische Abschätzung wird durch zweimaliges Ziehen einer Seite bei gleichzeitiger Reduktion der Rechentiefe realisiert. Es ist sehr wichtig, dass nie mehrere Nullzüge hintereinander ausgeführt werden, was eine Zusammenschrumpfung des Suchbaumens zur Folge hätte

```
if (NullzugOk()) {
  NullWert = -Suche(Farbwechsel(Farbe), Tiefe + 1, Distanz – NULLREDUKTION, -Beta, -Alpha);
  if (NullWert >= Beta) return Beta;
}
```

Für die Reduktion der Rechentiefe gilt allgemein

$$2 \leq \text{NULLREDUKTION} \leq 4.$$

Es haben sich noch weitere technische Verfeinerungen herauskristallisiert, die den Verzweigungsfaktor noch stärker reduzieren. Allgemein muss abgewogen werden, ob durch die Reduktion des Verzweigungsfaktors und die somit höheren Rechentiefen die heuristischen Fehlabschätzungen dieser Verfahren hinreichend kompensiert werden können.

4.2.2 Futility-Schnitte

Das *Wegschneiden* von Unterbäumen in der Nähe des Rechenhorizontes ist ebenfalls ein sehr kritisches Verfahren. Dieses Verfahren wird vorwiegend in der bereits entwickelten Ruhesuche eingesetzt

```
if (Positionsbewertung(Farbe) >= Beta) return Beta;
```

Auch kurz vor dem Suchhorizont arbeitet dieses Verfahren mit einer zusätzlichen Fehlerabschätzung sehr effizient und präzise

```
if  (Distanz <= 1) {
  if (Positionsbewertung(Farbe) >= Beta + FEHLERSCHWELLE) return Beta;
}
```

Für die Fehlerschwelle gilt allgemein

1 × Wert(Bauer) ≤ FEHLERSCHWELLE ≤ 5 × Wert(Bauer) ~ Wert(Turm).

Da die Funktionsweise dieses Verfahrens nur von der Unterschranke und Oberschranke abhängt, muss dieses Verfahren nicht unbedingt dem Nachfolgeknoten (Kindknoten) zugeordnet werden. Durch konsequente Anwendung des Nega-Max-Verfahrens (Tauschen von Unterschranke und Oberschranke und anschliessendes Negieren) folgt beispielsweise für Weiss:

```
if  (Distanz <= 1) {
  /* Positionsbewertung aus Sicht von Weiss */
  if (Positionsbewertung(WEISS) >= Beta + FEHLERSCHWELLE) { ... }
}
```

oder durch Anwendung des Nega-Max-Verfahrens folgt umgekehrt aus Sicht von Schwarz im übergeordneten Knoten (Vaterknoten)

```
→ if  (Distanz – 1 <= 1) {
     /* Positionsbewertung aus Sicht von Schwarz */
     if (Positionsbewertung(SCHWARZ) <= Alpha - FEHLERSCHWELLE) { ... }
   }
```

Beide Routinen arbeiten äquivalent und können je nach Konstruktion der Suchalgorithmen wahlweise eingesetzt werden.

4.2.3 Extensions

Einige Unterbäume werden wegen heuristischen Abschätzungen weggeschnitten. Allerdings sollten Unterbäume auch tiefer untersucht werden, wenn nach heuristischen Abschätzungen eine Zugfolge mit hoher Wahrscheinlichkeit einen Unterbaum produziert, der einen signifikanten Einfluss auf die zu berechnende Hauptvariante hat. Es haben sich einige Kriterien zur selektiven Erweiterung des Suchprozesses bisher sehr gut bewährt *(Ernst A. Heinz, Robert M. Hyatt):*

- der König wird bedroht
- ein Bauer rückt auf die vorletzte Reihe vor
- eine Schlagsequenz folgt
- eine *Zugzwangsituation* ist entstanden

4.3 Der Algorithmus zur Variantenberechnung

4.3.1 Die Variantensuche

Mit den bisher erarbeiteten Verfahren lässt sich bereits ein leistungsfähiger rekursiver Suchalgorithmus mit exponentieller Zeitkomplexität zur Variantenberechnung entwickeln.

```
int Variantensuche(int Farbe, int Tiefe, int Distanz, int Alpha, int Beta) {
  /* Weiss am Zug */
  if (Farbe == WEISS) {
    int HashWert, NullWert, Erweiterungen, Zug;
    /* Vor dem Suchhorizont -> Baumsuche */
    if (Distanz > 0) {
      /* Im Hashspeicher nach aktueller Stellung suchen, Kriterium Distanz beachten !! */
      if (StellungInHashspeicher(Distanz)) {
        HashWert = WertInHashspeicher();
        return HashWert;
      }
      /* Nullsuche ausführen (hier zieht Schwarz 2mal hintereinander), falls möglich */
      if (NullzugOk()) {
        NullWert = -Variantensuche(SCHWARZ, Tiefe + 1, Distanz –NULLREDUKTION, -Beta, -Alpha);
        /* heuristische Abschätzung war erfolgreich -> Unterbaum wegschneiden */
        if (NullWert >= Beta) return Beta;
      }
      /* Zugschleife durchlaufen */
      while ((Zug = NächsterZug())) {
        /* Zug ausführen, Variante berechnen, Zug zurücknehmen */
        ZugAusführen(Zug);
        /* Eventuell Suchbaum nach Zugausführung erweitern */
        Erweiterungen = ErweitereSuchbaum(Zug);
        /* Futility-Schnitt bei Folgezügen (nicht der erste Zug) und Distanz <= 1 */
        if (NichtErsterZug(Zug) && Distanz + Erweiterungen – 1 <= 1) {
          /* Position aus Sicht von „Farbe“ bewerten */
          if (Positionsbewertung(Farbe) <= Alpha - FEHLERSCHWELLE) {
            /* heuristische Abschätzung war erfolgreich -> Unterbaum wegschneiden */
            ZugZurücknehmen(Zug);
            continue;
          }
        }
        Alpha = Maximum(Alpha, -Variantensuche(SCHWARZ, Tiefe + 1,
                        Distanz + Erweiterungen – 1, -Beta, -Alpha));
        ZugZurücknehmen(Zug);
```

```
        /* Baumsuche abbrechen, wenn Unterschranke >= Oberschranke (Alpha-Beta-Pruning) */
        if (Alpha >= Beta) {
          /* Daten der zum Teil berechneten Stellung in Hashtabelle speichern */
          SpeichereDatenInHashtabelle(Distanz);
          return Beta;
        }
      }
      /* Daten der durchgerechneten Stellung in Hashtablle speichern */
      SpeichereDatenInHashtabelle(Distanz);
      return Alpha;
    }
    /* Suchhorizont erreicht -> Ruhesuche */
    else {
      int Wert, Zug;
      /* Position aus Sicht von „Farbe" bewerten */
      Wert = Positionsbewertung(Farbe);
      /* Futility-Pruning in der Ruhesuche */
      if (Wert >= Beta) return Beta;
      if (Wert > Alpha) Alpha = Wert;
      /* Zugschleife durchlaufen */
      while ((Zug = NächsterSchlagzug())) {
        /* Zug ausführen, Maximum der Stellung berechnen, Zug zurücknehmen */
        ZugAusführen(Zug);
        Alpha = Maxium(Alpha, -Variantensuche(SCHWARZ, Tiefe + 1, -Beta, -Alpha));
        ZugZurücknehmen(Zug);
        /* Ruhesuche abbrechen, wenn Unterschranke >= Oberschranke (Alpha-Beta-Pruning) */
        if (Alpha >= Beta) return Beta;
      }
      return Alpha;
    }
  }
  /* Schwarz am Zug (analog zu „Weiss am Zug", nur mit getauschten Farben rechnen) */
  else { ... }
}
```

Dieser Algorithmus ergibt sich durch Zusammenbauen der schon vorgestellten Suchalgorithmen **`Ruhesuche`** und **`Baumsuche`** erweitert um die Hashverfahren und die heuristischen Verfahren.

4.3.2 Rekonstruktion der Varianten

Eine *Hauptvariante* ist die von der Variantensuche berechnete optimalen Zugfolge. Eine Hauptvariante kann höchstens eine Zugfolge bis zum Suchhorizont verfolgen, Zugfolgen der Ruhesuche werden nicht berücksichtigt. Die berechnete Hauptvariante lässt sich nach vollendeter Suche durch rekursives Auslesen des Hashspeichers *(Stefan Zipproth)* sehr schnell rekonstruieren:

```
void HauptvarianteLesen(int Farbe, int Tiefe, int Zug, int Hauptvariante[MAX_TIEFE]) {
  /* Weiss am Zug */
  if (Farbe == WEISS) {
    int FolgeZug;
    /* Nächsten Zug der Hauptvariante eintragen */
    Hauptvariante[Tiefe] = Zug;
    /* Zug ausführen, nächsten HV-Zug aus Hashspeicher auslesen, Zug zurücknehmen */
    ZugAusführen(Zug);
    if ((FolgeZug = ZugInHashspeicher())) {
      if (Tiefe < MAX_TIEFE) {
        HauptvarianteLesen(SCHWARZ, Tiefe + 1, FolgeZug, Hauptvariante);
      }
    }
    ZugZurücknehmen(Zug);
  }
  /* Schwarz am Zug (analog zu „Weiss am Zug", nur mit getauschten Farben rechnen) */
  else { ... }
}
```

Dieser Beispielalgorithmus zur rekursiven Rekonstruktion der berechneten Hauptvariante ist analog zu den Suchverfahren, nur mit einem konstanten Verzweigungsfaktor von **1**. Hier wird nur die nach abgeschlossener Suche bekannte Hauptvariante vertieft. Einzig der Hashspeicher ist noch auszulesen. Es folgt daher eine konstante Zeitkomplexität

$$t_{hauptvariante\_lesen} \sim Verzweigungsfaktor^{Tiefe} = 1^{Tiefe} = 1$$

$$\rightarrow T_{hauptvariante\_lesen} = O(1) = konstant.$$

4.4 Vergleich zu den Matrixspielen

In nachfolgendem Matrixspiel minimiert der Spieler **B** den Erfolg von Spieler **A** nach dem Minimax-Prinzip durch Minimierung der *Spaltenmaxima*

Minimum(Maximum(55, 30, 75), Maximum(80, 35, 55), Maximum(10, 55, 55))
= Minimum(75, 80, 55) = 55.

	B1	B2	B3	Zeilenminima
A1	55	80	10	10
A2	30	35	55	30
A3	75	55	55	55
Spaltenmaxima	75	80	55	

Abbildung 4-2 Beispiel für ein Matrixspiel

Spieler **A** handelt entgegengesetzt und macht aus dem für ihn schlechtmöglichsten Fall das Beste durch Maximierung der *Zeilenminima*

Maximum(Minimum(55, 80, 10), Minimum(30, 35, 55), Minimum(75, 55, 55))
= Maximum(10, 30, 55) = 55.

Der Minimax-Prozess für die Variantensuche ist abgesehen von problemspezifischen Sonderverfahren ein rekursives *Matrixspiel*. Hierbei hätte die Matrix eine Grösse von

ZugmöglichkeitenWeiss × ZugmöglichkeitenSchwarz
~ DurchschnittlicheZugmöglichkeiten^2

	a7a5	...	h7h5	Zeilenminima
a2a4	2	...	7	Minimum(2, ..., 7)
...	...	...	...	...
h2h4	15	...	8	Minimum(15, ..., 8)
Spaltenmaxima	Maximum(2, ..., 15)	...	Maximum(7, ..., 8)	

Abbildung 4-3 Beispiel für Schachmatrix bei Rechentiefe 2

Diese Matrix ist ein Schachbaum der Tiefe 2. Durch weitere Vertiefungen würde jedem Eintrag der Matrix eine weitere Matrix rekursiv zugeordnet. Unter Vernachlässigung der problemspezifischen Lösungsverfahren für eine effiziente Variantensuche, kann das Minimax-Verfahren für die Variantensuche auch als rekursives Minmax-Verfahren für Matrixspiele bezeichnet werden.