Diplomarbeit

Johannes Hartz

Entwicklung eines Schachprogramms für ressourcenarme Systeme wie PDAs

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Johannes Hartz

Entwicklung eines Schachprogramms für ressourcenarme Systeme wie PDAs

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Softwaretechnik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Kai von Luck
Zweitgutachter : Prof. Dr. Christoph Klauck

Abgegeben am 20. Oktober 2005

Entwicklung eines Schachprogramms für ressourcenarme Systeme wie PDAs

Stichworte

ressourcenarme Systeme, mobile Systeme, PDA, Pocket PC, Computerschach, MinMax-Suche, PVS/NegaScout-Algorithmus, heuristische Bewertung

Zusammenfassung

Diese Diplomarbeit bewertet die Möglichkeiten eines ressourcenarmen Systems zur Implementation einer rechenintensiven Anwendung anhand eines hierfür entwickelten Schachprogramms. Es wird dargestellt, wie dieses Programm auf einem PDA möglichst effizient implementiert werden kann. Die theoretischen Grundlagen werden erläutert und es wird gezeigt, wie die praktische Umsetzung durch die Limitierungen der Zielplattform beeinflußt wird. Das Schachprogramm wird quantitativ und qualitativ evaluiert und es wird festgestellt, welche Performanz es erzielen kann. Dies ermöglicht eine Bewertung der Leistungsfähigkeit der Zielplattform sowie der verwendeten Verfahren. Außerdem wird dargestellt, welche möglichen Konsequenzen sich aus den Ergebnissen der Arbeit für die Zukunft ergeben.

Development of a chess program for low-resource systems like PDAs

Keywords

low-resource systems, mobile systems, PDA, Pocket PC, computer chess, MinMax search, PVS/NegaScout algorithm, heuristic evaluation

Abstract

This diploma thesis evaluates the capabilities of a low-resource system for the implementation of a computationally intensive application through the development of a chess program. It will be presented how this program can be implemented as efficiently as possible on a PDA. The theoretical basis will be explained and it will be shown how the realization is influenced by the limitations of the target platform. The chess program will be evaluated quantitatively and qualitatively and its performance will be determined. This makes an evaluation of the performance of the target platform and the applied techniques possible. Furthermore, it will be shown which possible consequences for the future derive from the results of the paper.

Danksagung

Ich möchte mich bei meiner Familie, meinen Mitbewohnern und meinen Freunden für die freundliche Unterstützung bedanken. Insbesondere bei Ludwig Hartz, Annette Zimmer, Klaas Neumann, Lars Burfeindt und Anna Franze.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Mobile Computer sind aus dem modernen Geschäftsleben nicht mehr wegzudenken. PDAs (Personal Digital Assistants) übernehmen heutzutage viele Aufgaben wie Datenverwaltung, Terminplanung, Internetzugriff usw..., die früher nur von stationären Systemen geleistet werden konnten. Der Faktor Mobilität hat allgemein in der Entwicklung moderner Mikroprozessortechnologie einen hohen Stellenwert erreicht: Es sind heutzutage tragbare Systeme auf dem Markt, die in ihrer Rechenleistung bereits den Personal Computern entsprechen, die Mitte der 90er Jahre verwendet wurden. Diese Geräte (die man auf Grund der durch ihre Mobilität anderweitig gegebenen Einschränkungen ressourcenarm nennen muß) sind in der Lage, komplexe Aufgaben auszuführen, da sie voll programmierbar und mit einem eigenen Betriebssystem ausgestattet sind. Mobile Systeme mit weit höherer Leistungsfähigkeit werden bereits entwickelt; der Markt für mobile Endgeräte wächst stetig, auch gefördert durch technische Entwicklungen wie WLAN oder Bluetooth, die eine starke Vernetzung dieser Geräte ermöglichen. Diese Entwicklung gilt ebenfalls für den Bereich der mobilen Telekommunikation: Die Geräte der neuen Handy-Generation, auch Smartphones genannt, werden mit einem eigenen Prozessor sowie einer Java VM (Virtuelle Maschine) ausgestattet, die es ermöglichen, eigenständige Programme auf das Gerät zu übertragen und auszuführen. Auch scheint sich ein Trend zu ergeben, die Fähigkeiten mehrerer Geräte in einer Plattform zu integrieren, wie es bei Handys mit eingebauter Digitalkamera der Fall ist. Andere mobile digitale Geräte, wie z.B. MP3-Player, haben eine ähnlich hohe Verbreitung erreicht wie Handys und werden ebenfalls mit diesen in einer Plattform integriert.
Insgesamt geht die technische Entwicklung in eine Richtung, die es vermuten läßt, daß in Zukunft mobile Endgeräte ein ständiger Begleiter der Menschen sein werden, was auch für mobile Computer gelten wird. Die Veränderung in der Verwendung von Computern, deren Beginn ich hier beschrieben habe, ist als „Ubiquitous Computing“ bekannt, und wird bereits wissenschaftlich analysiert [ETH Ubicomp]. Ein Ergebnis dieser Analyse ist, daß die Anzahl der Computer pro Nutzer immer weiter steigt, die Geräte jedoch gleichzeitig immer kleiner werden. Mark Weiser hat hierzu richtungsweisende Ideen hervorgebracht, u.a. in seiner Schrift „The Computer for the 21st Century“ [Weiser 91].

Gegenüber diesen modernen Entwicklungen ist das Schachspiel ein sehr altes Produkt der menschlichen Kreativität: Das erste schriftliche Zeugnis hierzu stammt aus der Zeit um 600 n.Chr. [Meyers 77]. Das Schachspiel ist wahrscheinlich in Indien entstanden, kam von dort nach Persien und verbreitete sich dann im ganzen arabischen Raum. Heute spielen Menschen auf der ganzen Welt Schach. Der Reiz des Spiels hat sich solange erhalten können, weil die fast unendliche Vielfalt möglicher Spielverläufe immer neue Ansätze und Variationen zur Gestaltung einer Partie bietet. Da im Schach kein Zufallselement vorhanden ist, hängt der Ausgang eines Partie ausschließlich von den Fähigkeiten der beteiligten Spieler ab, so daß der Gewinn eines Schachspiels oft als Ausdruck hoher geistiger Fähigkeiten verstanden wird: „Neben dem intellektuellen Reiz des Schachs ist der erzieherische Wert von Bedeutung. Schach lehrt Logik, Phantasie, Selbstdisziplin und Entschlossenheit.“ (Garry Kasparov)
Schon vor langer Zeit haben Menschen nun überlegt: Woher kommt das Schachspiel? Wer hat es erfunden? Da dies nicht genau bekannt ist, gibt es eine Reihe von Legenden, von denen ich eine hier wiedergeben möchte, sie stammt aus Persien und ist fast 800 Jahre alt [Schachlexikon 80]. Sie gibt auch darüber Auskunft, warum Schach heute immer noch genauso interessant ist wie schon vor vielen hundert Jahren.

<u>Die Legende vom Weizenkorn</u>
Nach einer alten arabischen Legende sollte der Weise, der das Schachspiel erfunden hatte, vom

König für seine Erfindung belohnt werden. Er verlangte, man lege ihm ein Weizenkorn auf das erste Feld eines Schachbretts, die doppelte Menge auf das Zweite, die doppelte Menge davon auf das dritte Feld usw... . Der König ließ sich darauf ein und erkannte das Problem nicht. Wenn man alle Weizenkörner zusammenzählt erhält man aufgrund des exponentiellen Wachstums:

$$2^{64} - 1 = 1.845 \times 10^{19} = 18446744073709551615 \text{ Weizenkörner}$$

Die Geschichte verdeutlicht das Außergewöhnliche des Schachspiels: Da die Menge der möglichen Spielverläufe sich nach den gleichen Gesetzen des exponentiellen Wachstums verhält, wie sie in der Geschichte deutlich werden, ist es klar, daß es nicht möglich ist, das Schachspiel an sich komplett zu analysieren. Es kann also selbst für den besten menschlichen Spieler oder den schnellstmöglichen Computer immer noch die Möglichkeit geben, eine Schachpartie zu verlieren, da der vollständige Ablauf eines Spiels mit analytischen Mitteln nicht zu erfassen ist. Schon viele Menschen waren begeistert von den Möglichkeiten, die sich dadurch eröffnen: „Ich wußte wohl aus eigener Erfahrung um die geheimnisvolle Attraktion des "königlichen Spiels", dieses einzigen unter allen Spielen, die der Mensch ersonnen, das sich souverän jeder Tyrannis des Zufalls entzieht und seine Siegespalmen einzig dem Geist oder vielmehr einer bestimmten Form geistiger Begabung zuteilt. Aber macht man sich nicht bereits einer beleidigenden Einschränkung schuldig, indem man Schach ein Spiel nennt? Ist es nicht auch eine Wissenschaft, eine Kunst [...]." [Zweig 74].
Und tatsächlich ist es so, daß Schach, gerade in der Moderne, auf sehr hohem wissenschaftlichen Niveau betrieben wird. So gibt es z.B. in Moskau eigene Universitäten, die sich mit der Ausbildung professioneller Schachspieler befassen und den Studenten in mehreren Jahren das vermitteln, was an Wissen über Schach bereits vorhanden ist. Es gibt ebenfalls eine außergewöhnlich große Menge an Literatur zum Thema Schach[1]; das Interesse am Schachspiel ist aufgrund der hohen Komplexität seines Ablaufs also immer noch ungebrochen.

Im 18. Jahrhundert kam erstmals die Frage auf, ob es möglich sei, das Schachspiel zu automatisieren. Hierzu wurden verschiedenste Maschinen konstruiert, die Schach spielen sollten. Sie waren dazu jedoch nicht selbständig in der Lage und konnten nur mit menschlicher Unterstützung eine komplette Schachpartie bestreiten [Schachmania]. Erst mit der Entwicklung der Computertechnologie hat sich dies geändert. Tatsächlich ist Schachprogrammierung eine der ältesten Anwendungen in der Informatik: Sie geht zurück bis in die 50er Jahre, in denen Claude Shannon [Shannon 49] und Allan Turing [Turing et al, 53] die ersten theoretischen Ansätze dessen entwickelt haben, was heute in jedem Schachcomputer zum Einsatz kommt. Bereits im Jahre 1958 ist der erste vollfunktionsfähige Schachcomputer entstanden, 1970 gab es die erste Meisterschaft für Schachprogramme und 1983 erreichte ein Schachprogramm zum ersten Mal den Rang eines internationalen Meisterspielers („Belle" von Ken Thompson und Joe Condon) [Wall].

Die „Legende vom Weizenkorn" gibt ebenfalls Aufschluß über die Problematik der Schachprogrammierung: Um Schach spielen zu können, muß ein Programm einen Spielbaum aufbauen, d.h. jeden möglichen Zug der aktuellen Stellung, die möglichen Folgezüge, die Züge, die darauf folgen etc... auswerten. Da die Menge der zu berechnenden Stellungen mit jedem Halbzug exponentiell wächst, ist es nicht möglich, alle möglichen Zugkombinationen zu berechnen. Der Spielbaum beim Schach ist in einer typischen Stellung von nicht zu bewältigender Größe: So wäre es notwendig, bei 30 möglichen Zügen pro Spieler (in einer typischen Mittelspielstellung) 650 Milliarden Stellungen auszuwerten, um 4 vollständige Züge zu analysieren ($30^8 = 656100000000$). Daraus ergibt sich als Folgerung für die Programmierung eines Schachcomputers:
Die in den Zugabfolgen eines Schachspiels erreichbare Komplexität ist nicht zu bewältigen, im Sinne der Findung des **optimalen** Zuges in einer ausgeglichenen Stellung. Die Herausforderung

1 Eine Suche bei amazon.de mit dem Suchwort „Schach" liefert 819 Treffer aus der Kategorie „Sport allgemein"!

besteht also darin, in endlicher Zeit eine möglichst gute **Annäherung** an diesen Zug zu finden. Die Optimierung dieser Annäherung ist es, die auch heute noch Programmierer moderner Schachprogramme zu erreichen versuchen.

Die grundlegenden Verfahren zur Lösung dieses Problems wurden in den 60er und 70er Jahren entwickelt. Der große rechnerische Aufwand hat es aber erst zum Ende der 90er Jahre möglich gemacht, daß ein Schachcomputer gegen den besten menschlichen Gegenspieler gewinnt: Im Mai 1997 fand in New York ein auf sechs Partien angelegter Wettkampf zwischen dem amtierenden Schachweltmeister Garry Kasparov und dem IBM-Großrechner Deep Blue statt. Deep Blue gewann mit 3,5 : 2,5 Punkten, nachdem Kasparov wenige Monate zuvor noch die Oberhand gegen den Computer behalten hatte. Erst seit diesem Sieg eines Schachcomputers gegen den amtierenden menschlichen Schachweltmeister, gilt das Gebiet der Schachprogrammierung als ein sowohl in der Theorie als auch in der Praxis verstandenes Problem.

Abb. 1.1: Garry Kasparov spielt gegen Deep Blue [MSN Encarta]

1.2. Zielsetzung und Aufbau der Arbeit

Ressourcenarme Systeme wie PDAs haben sich in der Praxis bereits bestens als Mittel zur Datenverwaltung, Terminplanung und Kommunikation bewährt. Im Bereich der rechenintensiven Anwendungen, d.h. Anwendungen, die höhere Leistungsfähigkeit durch höhere Qualität der Ergebnisse belohnen (z.B. Spiele, Streaming sowie alle Arten von Echtzeitanwendungen), sind diese Geräte hingegen bisher kaum analysiert worden. Ich werde in meiner Diplomarbeit die Fähigkeiten ressourcenarmer Systeme, am Beispiel eines PDA, auf diesem Gebiet erfassen und ihre Möglichkeiten bewerten. Schachprogrammierung ist zur Bewertung der Leistungsfähigkeit dieser Systeme sehr gut geeignet, da sie eine Vielzahl numerischer Verfahren beinhaltet: Die Ergebnisse der Optimierung dieser Verfahren können zeigen, inwieweit moderne mobile Geräte bereits in der Lage sind, rechenintensive Aufgaben angemessen zu bewältigen. Dies ist sowohl von theoretischem Interesse, im Sinne der durch das Ubiquitous Computing gegebenen Steigerung der Verbreitung mobiler Computer, als auch von praktischem, da die mobilen Systeme mit ihrer Vielzahl von Kommunikations- und Vernetzungsmöglichkeiten eine ideale Plattform für kommerzielle Spiele aller Art darstellen. Es erscheint außerdem reizvoll zu evaluieren, inwieweit die Leistungsfähigkeit moderner mobiler Geräte bereits derjenigen entspricht, die vor noch nicht allzulanger Zeit für dedizierte Schachcomputer zur Verfügung stand; dies gibt ebenfalls Aufschluß über ihre Möglichkeiten. Anhand dieser Ergebnisse kann man außerdem versuchen zu prognostizieren, inwieweit mobile Geräte in Zukunft stationäre Systeme ersetzen könnten.

Um die dargestellten Bewertungen durchführen zu können, werde ich ein Schachprogramm namens Nemesis[2] für einen Hewlett Packard iPAQ H5500 PDA, der mir freundlicherweise von der HAW-Hamburg zur Verfügung gestellt wurde, entwickeln. Die Zielsetzung hierbei ist, daß Nemesis 1.0 auf dem Niveau eines guten Vereinsspielers Schach spielen soll. Sollte dieses Ziel erreicht werden, so wäre die Umsetzung eines Schachprogramms auf einem PDA zusätzlich von praktischem Nutzen, da dieses Programm für einen durchschnittlichen Anwender ein interessantes Produkt mit einer langfristigen Herausforderung darstellen würde.

Im theoretischen Teil der Arbeit werde ich zuerst verschiedene Grundlagen der Schachprogrammierung erläutern. Hierbei beziehe ich mich ausschließlich auf Ansätze, die in Nemesis implementiert wurden, da sich im Laufe der Entwicklung herausgestellt hat, daß sie unverzichtbar sind. Bei der Darstellung der verschiedenen Verfahren konzentriere ich mich ebenfalls auf diejenigen, die in Nemesis implementiert wurden. Zur praktischen Umsetzung einiger Ansätze gibt es jedoch verschiedene Möglichkeiten, die deshalb in diesem Kapitel als Alternativen aufgezeigt werden. Später werde ich im Abschnitt „Auswahl der Verfahren“ erläutern, welche der Alternativen, im Hinblick auf die Möglichkeiten und Einschränkungen der Zielplattform, ausgewählt wurden, und welche spezifischen Gründe es hierfür gibt.
Zu Beginn des praktischen Teils der Arbeit wird dargestellt, welche Plattformen in Vergangenheit und Gegenwart für Schachprogramme genutzt wurden und welche in Zukunft zur Verfügung stehen könnten. Die technischen Spezifikationen und Leistungsmerkmale des verwendeten PDAs werden vorgestellt und es wird gezeigt, welche Rahmenbedingungen durch die Verwendung dieser Plattform gegeben sind. Im weiteren Verlauf dieses Abschnitts werde ich zeigen, warum einzelne Verfahren deutlich besser für die Implementation einer Schachanwendung in ressourcenarmen Systemen geeignet sind als andere. Außerdem werde ich darstellen, wie die ausgewählten Verfahren unter den gegebenen Bedingungen implementiert wurden, und welche Details es dabei zu beachten gilt.
Nachdem die Implementation von Nemesis vorgestellt wurde, werde ich die Leistungsfähigkeit der entwickelten Anwendung sowohl quantitativ als auch qualitativ testen und bewerten. Ich werde analysieren, welche der verschiedenen Verfahren aufgrund der begrenzten Ressourcen der

2 Rachegöttin der griechischen Mythologie

verwendeten Plattform die größten Optimierungen erzielen können. Darauf aufbauend werde ich die Tauglichkeit eines PDAs für eine praxisorientierte Schachanwendung bewerten. Der Vergleich der Leistungsfähigkeit von Nemesis auf einem PDA und auf einem PC läßt weitere Rückschlüsse auf die Möglichkeiten eines ressourcenarmen Systems zur Ausführung einer rechenintensiven Anwendungen zu. Ein weiteres Ziel der Evaluation ist, herauszufinden, welche Spielstärke Nemesis 1.0 auf dem Hewlett Packard iPAQ H5500 PDA erreichen kann. Diese werde ich ebenfalls quantitativ und qualitativ bewerten. Außerdem werde ich zeigen, welche Konsequenzen, aufgrund der weiteren technischen Entwicklung, sich aus den Ergebnissen der Arbeit für die Zukunft ergeben könnten.

2. Theoretische Einführung

2.1. Grundlagen der Schachprogrammierung

2.1.1. MinMax-Suchalgorithmen

Die künstliche Intelligenz eines Schachprogramms basiert auf der Bewertung der in einer gegebenen Stellung möglichen Züge. Hierzu wird der MinMax-Rückbewertungsalgorithmus verwendet, der für alle Spiele der Klasse endliches Zweispieler-Nullsummenspiel mit vollständiger Information anwendbar ist [Neumann v.; Morgenstern 44]. Der theoretische Ablauf des Algorithmus' ist folgender: Der Spieler, der gerade am Zug ist, testet einen seiner möglichen Züge, danach spielt er anstelle seines Gegners und macht auf dem simulierten Spielfeld einen der Züge, die für diesen Spieler möglich sind. Dies wechselt sich solange ab, bis entweder ein Spielende erreicht ist oder ein gegebenes Zug- oder Zeitlimit überschritten wurde. Wenn ein Spieler diesen Punkt erreicht hat, führt er keine weiteren Spielzüge mehr aus, stattdessen berechnet er eine Bewertung der gefundenen Stellung. Dieses Ergebnis wird an die Vorgängerposition zurückgegeben. Sind alle möglichen Folgezüge dieser Position ausgewertet, wird angenommen, daß der dort aktive Spieler den Zug mit dem höchsten errechneten Ergebnis wählen wird. Das Ergebnis dieses Zuges wird eine Ebene in der Suche zurückgereicht, dort werden wiederum alle weiteren möglichen Züge ausgeführt und das beste Ergebnis (für den anderen Spieler) wird ermittelt. Dieses Verfahren erfolgt rückbewertend für jede Ebene bis zum Ausgang der Suche. Nachdem also ein möglicher **Suchpfad** analysiert wurde, geht man die Züge Schritt für Schritt zurück und führt stattdessen andere Züge aus, so daß alle möglichen Züge jeder untersuchten Stellung berücksichtigt werden. Am Ausgangspunkt der Suche ist dann eine optimale Zugwahl[3], bei gleichzeitig angenommener optimaler Spielweise des Gegenspielers, möglich.
Für die praktische Anwendung des Verfahrens werden die möglichen Züge eines Spielers sowie die Folgezüge des Gegenspielers in einem Und/Oder-Baum dargestellt. Jeder Knoten in diesem **Suchbaum** entspricht einer Spielposition, die aktuelle Spielposition ist die Wurzel des Baumes. Die Bewertung eines Knotens wird anhand der Söhne bestimmt, wobei Blätter einen festen Wert besitzen (s. 2.1.10.), da sie Endsituationen des Spiels darstellen. Es wird festgelegt, daß positive Bewertungen gute Spielzüge für Spieler A und negative Bewertungen gute Züge für Spieler B sind[4], die Bewertung erfolgt also symmetrisch zur Null. Wenn die Tiefe des Baumes zu groß werden sollte, wird ab einer bestimmten **Suchtiefe** die Suche nicht fortgeführt. Da die Knoten an diesem **Suchhorizont** kein definiertes Spielende darstellen, gibt es auch kein festes Maß für ihre Bewertung. In einer praktischen Anwendung des MinMax-Verfahrens wird die Bewertung dieser Knoten durch eine Bewertungsfunktion (4.5.5.) heuristisch angenähert.
Der Baum wird nach Art einer Tiefensuche traversiert, dies geschieht durch den rekursiven Aufruf einer Suchfunktion. Die Suche erfolgt solange, bis ein Blatt oder ein anderer durch Begrenzung der Suchtiefe erzwungener Endknoten gefunden wird. Dessen Vater bekommt eine Bewertung, die dem bestmöglichen Ergebnis entspricht, das durch Zugwahl des aktiven Spielers an diesem Knoten erreicht werden kann. Dies hängt davon ab, ob Spieler A (der versucht, das Ergebnis zu maximieren) oder Spieler B (der versucht, das Ergebnis zu minimieren) den Zug gemacht hat. Für Spieler A wählt man den Zug mit dem größten Wert, da dieser ein gutes Ergebnis für ihn darstellt. Wenn aber Spieler B den letzten Zug gemacht hat, muss der Zug mit der kleinsten Zahl gewählt werden, da er für den minimierenden Spieler das beste Ergebnis ist. Auf diese Weise werden nun alle Suchebenen nach obigem Schema maximierend oder minimierend rückbewertet. Wenn alle Züge an der Wurzel ausgewertet wurden, kann der aktive Spieler den Zug mit dem bestmöglichen Ergebnis auswählen.

3 Bezogen auf die Menge der Informationen, die aus der MinMax-Suche bekannt ist
4 Das grundlegende Kriterium eines Nullsummenspiels

Man kann dieses Verfahren am besten an einem einfachen Beispiel erläutern:

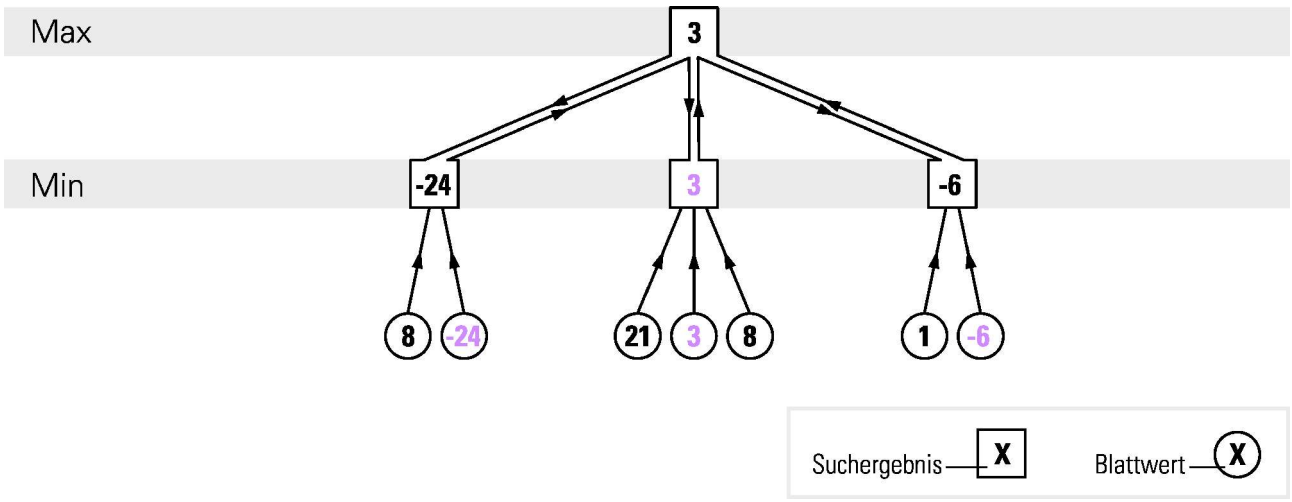


Abb. 2.1: Ablauf einer MinMax - Suche[5]

Es erfolgt hier eine Suche zur Tiefe zwei (dann ist der Suchhorizont erreicht und die dargestellten Ergebnisse werden von der Bewertungsfunktion errechnet). Im Ablauf der Tiefensuche wird zuerst das Blatt mit dem Wert 8 erreicht. Dieser Wert wird zurückgegeben, danach wird das zweite Blatt mit dem Wert -24 expandiert. Der minimierende Spieler wählt nun das für ihn bestmögliche Ergebnis, in diesem Fall -24. Dieser Wert wird zur Wurzel zurückgegeben. Der maximimierende Spieler expandiert nun seine weiteren Zugmöglichkeiten, die als mögliche Ergebnisse 3 und -6 haben, sein bestmögliches Ergebnis an der Wurzel ist also 3. Der Zug, der zu diesem Ergebnis geführt hat, würde vom maximierenden Spieler als bester Zug gewählt werden.

In der Praxis wird die MinMax-Suche meist als NegaMax-Variante implementiert:

```
int NegaMax(Stellung s, int tiefe) {

   if (tiefe == 0 OR Blatt(s)) RETURN Bewertung(s);

   int best = -INFINITY;

   Finde Nachfolger s_1,...,s_w von s;

   for (int i = 1; i <= w; i++) {

      int wert = -NegaMax(s_i, tiefe-1);

      if (wert > best) best = wert;
   }

  return (best);
}
```

Diese Variante geht ausschließlich maximierend vor und unterzieht hierfür die Ergebnisse bei ihrer Weitergabe einem Vorzeichenwechsel. Dies ist deshalb möglich, da die Evaluation symmetrisch zur Null erfolgt, das Verhalten des Algorithmus' ändert sich dabei gegenüber dem beschriebenen Ablauf der MinMax-Suche nicht. Man verwendet diese Variante vor allem deswegen, weil sie eleganter und im Ablauf etwas schneller ist, sie ist jedoch intuitiv schwieriger nachzuvollziehen. Für weiterführende Literatur zu diesem Thema verweise ich auf [Ginsberg 93].

5 Abbildungen ohne Quellenangabe wurden vom Verfasser generiert

2.1.2. Reduzierung des Suchraums durch Alpha/Beta-Schnittverfahren

Da, wie in der Einleitung dargestellt, der Suchraum zu groß ist, um ihn in angemessener Zeit komplett zu analysieren, hat man lange nach Verfahren gesucht, diesen Suchraum zu verkleinern, um die MinMax-Suche zu beschleunigen. Eine Möglichkeit hierzu ist das Alpha/Beta-Schnittverfahren. Anfangs nur als heuristisches Verfahren gedacht, ist mittlerweile bewiesen, daß Alpha/Beta-Schnitte den Suchraum **ohne Informationsverlust**, d.h. ohne Beeinflussung des Suchergebnisses, verkleinern können [Brudno 63 / Knuth; Moore 75].
Die Idee des Alpha/Beta-Verfahrens ist, daß zwei Werte, Alpha und Beta, weitergereicht werden, die ein minimal bzw. maximal mögliches Ergebnis der Spieler beschreiben. Der Alpha-Wert ist das Ergebnis, das der maximierende Spieler mindestens erreichen kann, der Beta-Wert ist das Ergebnis, das der minimierende Spieler mindestens erreichen kann. Die Alpha/Beta-Werte stellen also **Referenzwerte** auf bereits bekannte Suchergebnisse dar. Der Wertebereich, der durch Alpha und Beta begrenzt wird, wird **Suchfenster** genannt.
Besitzt ein maximierender Knoten eine Zugmöglichkeit, deren Ergebnis den Beta-Wert einstellt oder überschreitet, so wird die Suche an diesem Knoten abgebrochen, ein Beta-Schnitt (englisch: Beta Cutoff) findet statt. Der abgeschnittene Unterbaum muß nicht untersucht werden, da sein Ergebnis keinen Einfluß auf das Gesamtergebnis haben kann. Dies ist gegeben, da diese Variante den Mindestgewinn des minimierenden Spielers unterschreitet und er deshalb nicht zulassen würde, daß sie gespielt wird. Liefert ein Zug des maximierenden Spielers stattdessen ein Ergebnis, das den Alpha-Wert (also seinen Mindestgewinn) übersteigt, wird dieser Wert entsprechend nach oben angehoben. Analog gilt dies für die minimierenden Knoten, wobei für Werte kleiner gleich Alpha abgebrochen wird und für Werte kleiner Beta dieses nach unten angepasst werden kann.
In der NegaMax-Variante werden die Alpha/Beta-Werte für die nächste Suchebene sowohl negiert als auch getauscht. Dies hat keine Auswirkung auf den beschriebenen Ablauf, ist aber notwendig, da in dieser Variante beide Spieler maximierend vorgehen. Aus Sicht des aktiven Spieles beschreibt Alpha hier eine **Untergrenze** und Beta eine **Obergrenze** für das mögliche Ergebnis.

```
int NegaMaxAlphaBeta(Stellung s, int tiefe, int alpha, int beta) {

   if (tiefe == 0 OR Blatt(s)) RETURN Bewertung(s);

   int best = -INFINITY;

   Finde Nachfolger s_1,...,s_w von s;

   for (int i = 1; i <= w; i++) {

      if (best > alpha) alpha = best;               //Alpha - Anpassung

      int wert = -NegaMaxAlphaBeta(s_i, tiefe-1, -beta, -alpha);

      if (wert > best) best = wert;

      if (best >= beta) return (best);              //Beta - Schnitt
   }

  return (best);
}
```

Die folgende Abbildung zeigt die Suche in einem Beispielbaum mit 28 Knoten, von denen nur 21 ausgewertet werden müssen, um das Ergebnis zu finden, die anderen 7 können ohne Informationsverlust abgeschnitten werden:

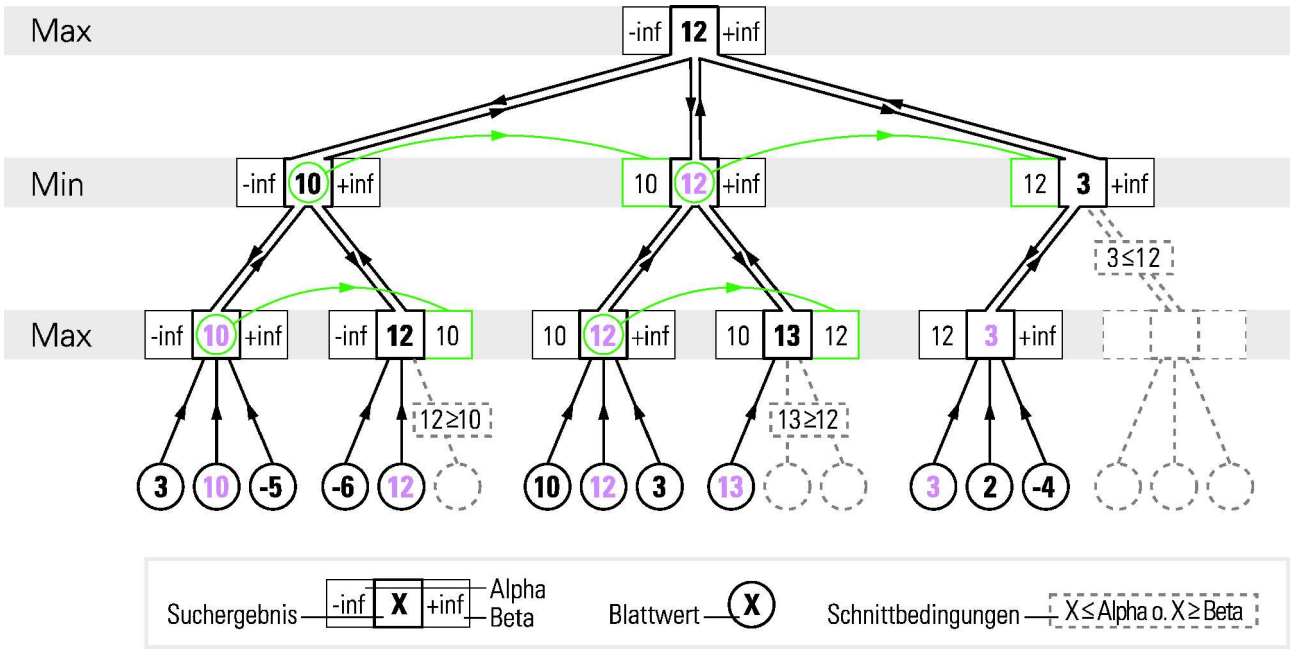


Abb. 2.2: Ablauf einer Alpha/Beta - Suche

Der Suchalgorithmus verwendet ein Alpha/Beta-Suchfenster, der Alpha-Wert stellt dabei die untere Grenze und der Beta-Wert die obere Grenze des Fenster dar. Dieses Fenster wird an jeden Kindknoten weitergegeben, wobei an der Wurzel mit einem maximalen Fenster, also den Werten minus unendlich und plus unendlich (dargestellt durch -inf und +inf), begonnen wird. Die ersten drei Blätter werden von einem maximierenden Knoten ausgewertet, der als besten Wert 10 erreicht. Der Beta-Wert kann nun angepasst werden (verdeutlicht durch den grünen Pfeil), und das neue Suchfenster [-unendlich, 10] wird dem nächsten maximierenden Knoten übergeben. Der Blattwert 12, der hier erreicht werden kann, ist so gut, daß er den Beta-Wert 10 überschreitet, somit müssen weitere Blätter nicht mehr betrachtet werden: Das Ergebnis 12 ist für den maximierenden Spieler besser als das des linken Teilbaumes, deshalb würde diese Variante vom minimierenden Spieler (der die Auswahl zwischen diesen beiden Möglichkeiten hat) nicht zugelassen werden. Der gleiche Ablauf ergibt sich bei dem Beta-Schnitt mit der Bedingung 13 ≥ 12. Aus dem linken Teilbaum hat der maximierende Spieler bereits das Ergebnis 12 erreicht, da er aber im rechten Teilbaum ein noch höheres Ergebnis erreichen kann, wird diese Variante vom minimierenden Spieler verhindert. Analog verhält es sich beim minimierenden Knoten mit dem Alpha-Schnitt gegen den Wert 12: Obwohl dieser Teilbaum erst teilweise ausgewertet wurde, ist klar, daß der maximierende Wurzelknoten diese Variante niemals wählen würde, weil der minimierende Spieler hier mindestens das Ergebnis 3 erzwingen könnte, während für den maximierenden Spieler aus dem mittleren Teilbaum das Ergebnis 12 sichergestellt ist.

Alpha/Beta-Schnittverfahren haben einen Seiteneffekt, der sich für tatsächliche Anwendungen in Spielen als problematisch herausstellt: Falls alle Ergebnisse eines Knotens kleiner gleich Alpha sind (dies wird Alpha Fail genannt), so bedeutet das, daß jeder seiner Kindknoten einen Beta-Schnitt verursacht hat. Es ist also klar, daß keiner der Unterbäume ein endgültiges Ergebnis geliefert hat, da keiner von ihnen vollständig untersucht wurde: Alle errechneten Ergebnisse basieren auf Schnittwerten, die lediglich eine Verletzung des Suchfensters beweisen können. Das

Ergebnis an diesem Knoten beschreibt eine Verletzung der Untergrenze dieses Suchfensters, die Suche hat also Alpha als Obergrenze der möglichen Ergebnisse gefunden. Es gibt jedoch keine Information darüber, welcher Kindknoten (also welcher der möglichen Züge) das tatsächlich **bestmögliche** Ergebnis erreichen kann, da aus keinem der Unterbäume ein exakter MinMax-Wert bekannt ist. Die Suche hat keinen eigenen Referenzwert erzeugt, der Vergleich der verschiedenen Obergrenzen aber kann keine gültige Information über den bestmöglichen Zug enthalten. Dieser Fall ist immer dann problematisch, wenn der beste Zug für die Anwendung eines Verfahrens zur Vorsortierung gespeichert werden soll (s. 2.1.8. und 2.2.7.), dies ist an diesen Knoten nicht möglich. Die gleiche Problematik gilt in noch stärkerem Maße für die Anwendung eines Aspiration Window (s.u.) mit einem möglichen Alpha Fail an der Wurzel.

2.1.3. Reduzierung des Suchraums durch Aspiration Search

Je kleiner das Suchfenster ist, desto mehr Knoten können in einer Alpha/Beta-Suche abgeschnitten werden. Das anfänglich mit minus unendlich und plus unendlich initialisierte Suchfenster schrumpft automatisch mit jeder neu berechneten Zugvariante. Um den langsamen Schrumpfungsprozeß zu beschleunigen, kann man die Suche aber auch sogleich mit einem **reduzierten Suchfenster** [*n* - epsilon, *n* + epsilon] beginnen. Dieses Suchfenster ist um einen vorab geschätzten MinMax-Wert *n* angeordnet und wird **Aspiration Window** genannt. In einer Aspiration Search wird das Suchfenster an der Wurzel also nur in einem begrenzten Bereich um *n* herum geöffnet, so daß mehr Schnitte im Suchbaum verursacht werden können; eine Grundannahme hierbei ist, daß das Ergebnis der Suche in einem ähnlichen Bereich wie der Schätzwert *n* liegen wird.

Eine Aspiration Search hat drei mögliche Ausgänge:

1. Das Ergebnis der Suche liegt innerhalb des Suchfensters. Die Suche war erfolgreich und hat mit reduziertem Aufwand (gegenüber einer einfachen Alpha/Beta-Suche) das richtige Ergebnis gefunden.
2. Das Ergebnis ist größer oder gleich Beta. Die Suche hat dadurch eine Untergrenze auf das tatsächliche Ergebnis bewiesen (da Beta den Mindestgewinn des Gegenspielers repräsentiert). Die Suche muß nun mit einem zwischen diesem Ergebnis und positiv unendlich geöffneten Suchfenster wiederholt werden, um den exakten Ergebniswert zu finden.
3. Das Ergebnis ist kleiner oder gleich Alpha. Die Suche hat eine Obergrenze auf das tatsächliche Ergebnis bewiesen. Die Suche muß nun mit einem zwischen negativ unendlich und dem vorherigen Ergebnis geöffneten Suchfenster wiederholt werden.

Sollte ein exaktes Ergebnis gefunden werden, also ein Ergebnis, das innerhalb des Suchfensters liegt, so war die Aspiration Search erfolgreich. Da durch das reduzierte Suchfenster deutlich mehr Knoten abgeschnitten werden konnten, konnte die Effizienz gegenüber einem einfachen Alpha/Beta-Verfahren deutlich verbessert werden.

Bei einem Suchergebnis, das größer gleich Beta ist, ist klar, daß sich der aktive Spieler verbessern wird, da ja eine neue Untergrenze auf das tatsächliche Ergebnis gefunden wurde. Selbst wenn die **Wiederholungssuche**, zur Bestimmung dieses Ergebnisses, aus Zeitgründen nicht mehr durchgeführt werden kann, so ist doch klar, welche Zugmöglichkeit zu dem besten Ergebnis geführt hat. Dieser Zug kann nun gewählt werden, denn auch wenn das exakte Ergebnis nicht bekannt ist, so ist doch bewiesen, daß dieser Zug zu einer Verbesserung des Suchergebnisses über die vorher angenommene Obergrenze des Suchfensters geführt hat, der Zug also eine Ergebnisverbesserung größer gleich der Hälfte der Größe des Aspiration Windows darstellt.

Der Fall eines Alpha Fail an der Wurzel ist in einer praktischen Anwendung gefährlich, da hier eine Obergrenze auf das Ergebnis bewiesen wurde. Es ist also nur klar, daß das zu erreichende Ergebnis kleiner oder gleich Alpha ist, es ist jedoch nicht klar, welcher der beste Zug in dieser Stellung ist, da ein Alpha Fail diese Information nicht ermitteln kann (s. 2.1.2.). Es ist nur bekannt, daß sich das Ergebnis um einen Wert größer gleich der Hälfte der Fenstergröße **verschlechtern** wird. Das Ergebnis der Suche ist solange wertlos, bis eine Wiederholungssuche durchgeführt wurde, die den Zug mit dem bestmöglichen Ergebnis ermitteln kann. Dies geschieht durch eine Alpha/Beta-Suche mit einem nach unten geöffneten Suchfenster, wodurch ein weiterer Alpha Fail ausgeschlossen wird. Dieser Fall ist in einer praktischen Anwendung deswegen besonders problematisch, da Computerschachspiele immer innerhalb eines Zeitlimits ausgetragen werden. Gerade bei hohen Suchtiefen, die zu hohen Suchzeiten führen, ist nun eine Suche, die kein verwertbares Ergebnis liefert, äußerst unerwünscht, da unter Umständen nur sehr gering verfügbare Zeit verschwendet wurde. Außerdem **muß** der zusätzliche Aufwand der Wiederholungssuche unbedingt geleistet werden, um einen guten Zug zu finden, der ausgespielt werden kann.

Trotz der dargestellten Problematik findet Aspiration Search allgemeine Anwendung (bei Suchverfahren, die mit geöffnetem Suchfenster beginnen), da der Nachteil, der durch Wiederholungssuchen entstehen kann, deutlich durch den Performanzgewinn ausgeglichen wird, der durch mehr Schnitte im Suchbaum möglich ist.

```
int AspirationWindowNegaMaxAlphaBeta(Stellung s, int tiefe, int n) {

   int epsilon;                 //Fenstergröße

   int alpha = n - epsilon;     //begrenztes Suchfenster um den Wert n
   int beta = n + epsilon;

   int wert = NegaMaxAlphaBeta(s, tiefe, alpha, beta);

                                //Wiederholungssuche?
   if (wert >= beta)
      wert = NegaMaxAlphaBeta(s, tiefe, wert, +INFINITY);

   else if (wert <= alpha)
      wert = NegaMaxAlphaBeta(s, tiefe, -INFINITY, wert);

   return (wert);
}
```

2.1.4. Reduzierung des Suchraums durch Nullfenstersuche

Eine Nullfenstersuche expandiert den Suchbaum mit einem extrem reduzierten Suchfenster, das **geschlossenes Suchfenster** oder Nullfenster genannt wird. Hierbei handelt es sich um ein leeres Suchintervall $[n, n + 1]$, das kein Element enthält, vorausgesetzt der Wertebereich ist ganzzahlig. Eine einmalige Anwendung einer Nullfenstersuche ist nicht zur Berechnung des MinMax-Wertes geeignet, da dieser stets außerhalb des Suchfensters liegt, also nicht exakt ermittelt werden kann. Das Ergebnis der Suche ist nur in der Lage, eine Grenze auf das tatsächliche Ergebnis zu beweisen, da jede Suche ein Ergebnis kleiner gleich Alpha (also eine Obergrenze) oder aber größer gleich Beta (also eine Untergrenze) liefern muß. Es kann also mit Hilfe einer Nullfenstersuche nur geprüft werden, ob der MinMax-Wert eines Suchbaums unter oder oberhalb eines gegebenen Referenzwertes n liegt.

Zur Veranschaulichung werde ich dieses Vorgehen nun an einem Beispiel verdeutlichen.

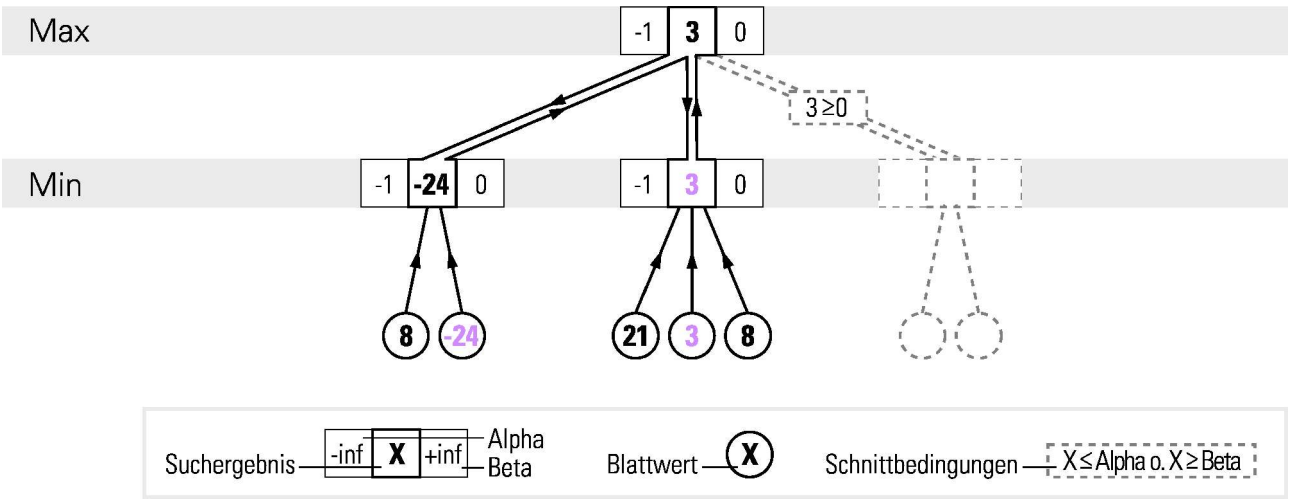


Abb. 2.3: Ablauf einer Nullfenstersuche

Das Nullfenster wird hier durch die Werte -1 und 0 beschrieben, die Suche ermittelt also, ob das Ergebnis dieses Baums entweder größer gleich oder kleiner dem Wert 0 ist (aus Sicht des maximierenden Spielers, für den ein Beta-Schnitt eine Untergrenze bedeutet). Die beiden linken Teilbäume lassen keine Schnitte zu, da die Ergebnisse des minimierenden Spielers entweder nicht kleiner gleich -1 sind, oder aber keine weiteren Blätter folgen, die abgeschnitten werden könnten. Der rechte Teilbaum hingegen kann komplett von der Suche ausgeschlossen werden, da ein Beta-Schnitt des maximierenden Spielers möglich ist: Durch die Ergebnisse der expandierten Knoten ist bereits sichergestellt, daß das Suchergebnis für ihn mindestens 3 beträgt, die Nullfenstersuche hat diesen Wert also bereits als eine Untergrenze des Suchbaums beweisen können.

Der Vorteil dieser Variante der Alpha/Beta-Suche liegt darin, daß durch das leere Suchintervall kein Ergebnis innerhalb des Fensters liegen kann, die Nullfenstersuche also die durch Anwendung eines Suchfensters **maximal** mögliche Menge an Alpha/Beta-Schnitten im Suchbaum erreicht. Die Geschwindigkeit der MinMax-Suche wird erhöht, da der ausschließliche Beweis von Grenzen mit deutlich weniger Aufwand geleistet werden kann. Aus dem gleichen Grund ist eine Nullfenstersuche aber auch nicht in der Lage, absolute MinMax-Werte zu liefern: Sie verwendet ausschließlich **relative Ergebnisse**, da die Grenzen des geschlossenen Suchfensters an jedem Knoten als Referenzwerte benutzt werden. Diese Referenzwerte werden aber nicht von der Suche selber ermittelt, sondern im vorhinein willkürlich bestimmt. Dies führt dazu, daß das Suchergebnis nur relativ zu den Grenzen des Nullfensters verstanden werden kann, was seine Verwertbarkeit deutlich einschränkt.

Es gibt dennoch verschiedene Möglichkeiten, dieses Verfahren innerhalb einer MinMax-Suche zu verwenden: Eine Möglichkeit ist, eine Nullfenstersuche zu benutzen, um für bestimmte Unterbäume eine Obergrenze zu beweisen und so sicherzustellen, daß sie keinen Einfluß auf das Suchergebnis haben können, da ihr Ergebnis, im Vergleich zu einem bekannten Referenzwert, zu schlecht ist. In dieser Art werden Nullfenstersuchen im Ablauf des **PVS/NegaScout-Algorithmus** verwendet (2.2.1.).
Eine weitere Möglichkeit ist, durch wiederholte Anwendung einer Nullfenstersuche den MinMax-Wert des Suchbaumes einzukreisen und so zu berechnen. Hierzu werden wiederholte Suchen zur gleichen Tiefe durchgeführt, die Ober- und Untergrenzen auf das Ergebnis zurückliefern. Diese Grenzen werden sukzessive angenähert, bis der exakte MinMax-Wert ermittelt ist, der **MTD(f)-Algorithmus** (2.2.2) entspricht dieser Anwendung des Verfahrens.

2.1.5. Reihenfolge der Knotenexpansionen

Die Reihenfolge der Knotenexpansionenen ist sehr wichtig für die Effizienz von Schnittverfahren und ihre deutliche Überlegenheit gegenüber der einfachen MinMax-Suche. Da im Ablauf der Alpha/Beta-Suche immer zuerst Blattwerte als Referenzwerte für Alpha- oder Beta-Schnitte ermittelt werden müssen, können am meisten Schnitte im Suchbaum vorgenommen werden, wenn diese Werte möglichst hoch sind, d.h. Knoten mit hohem Ergebnis **frühzeitig** expandiert wurden. Dies kann man gut an dem gleichen Suchbaum nachvollziehen, der bereits zur Darstellung der Alpha/Beta-Suche verwendet wurde:

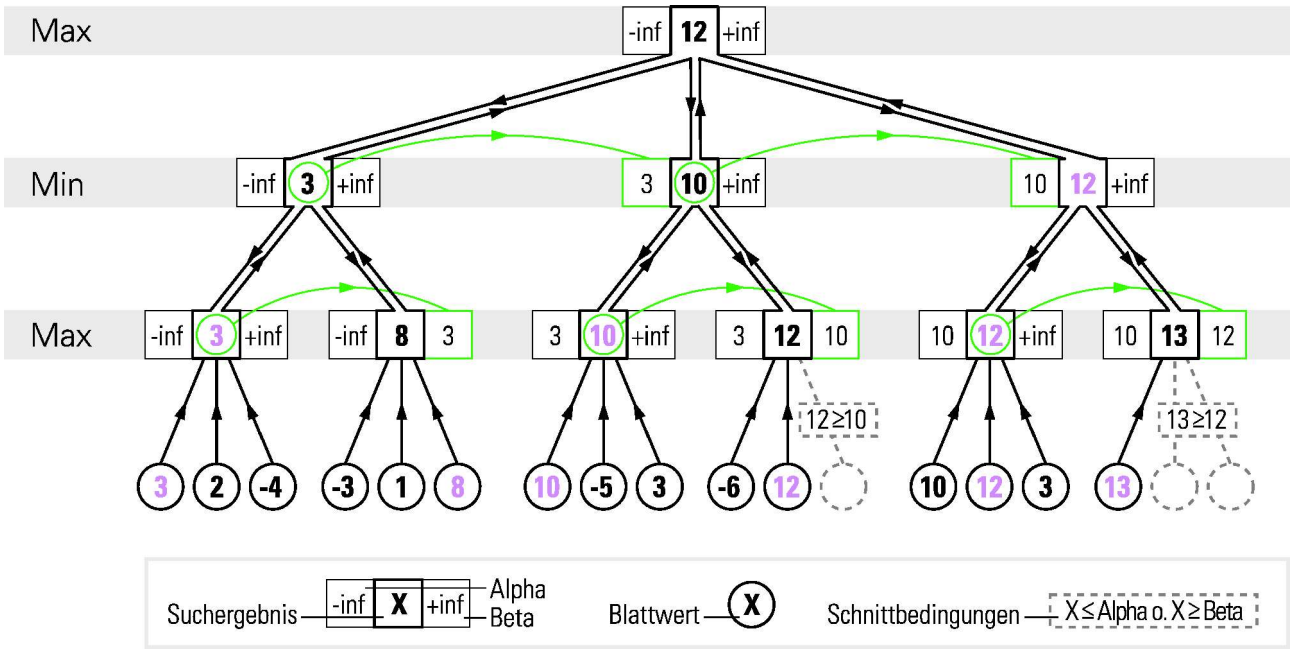


Abb. 2.4: Auswirkungen einer geänderten Reihenfolge der Knotenexpansionen

Der Alpha-Schnitt, der es in Abbildung 2.2 ermöglicht hat, den rechten Unterbaum mit dem Teilergebnis 3 gegen den Alpha-Wert 12 abzuschneiden, kann nur erfolgen, weil dieser nach einem anderen expandiert wurde, der Alpha auf einen Wert größer gleich 3 angehoben hat. Wird dieser Unterbaum bereits als erster untersucht, wie in Abbildung 2.4 zu sehen, so gibt es dort keine Schnitte, da keine genügend hohen Alpha- und Beta-Wert vorliegen, um sie zu verursachen. Die in Abbildung 2.2 dargestellten Schnitte sind zum Teil nur deswegen möglich, da frühzeitig hohe Referenzwerte erzeugt wurden. Insgesamt reduziert sich die Menge der in diesem Baum möglichen Schnitte bei hier der dargestellten Änderung der Reihenfolge der Knotenexpansionen bereits um über 50%.
Dieses Beispiel zeigt, wie wichtig die Reihenfolge der möglichen Züge für den Effizienzgewinn durch Alpha/Beta-Schnittverfahren ist: Züge mit hohen Ergebnissen müssen früh expandiert werden, um möglichst viele Schnitte im Suchbaum erzeugen zu können. Außerdem kann durch frühe Erzeugung hoher Ergebnisse die Anzahl der notwendigen Wiederholungssuchen sowohl für den PVS/NegaScout – Algorithmus als auch für den MTD(f) – Algorithmus verringert werden (s. 2.2.1. und 2.2.2.); dies führt ebenfalls zu einer Steigerung der Performanz.
Diesem Problem wird in der Schachprogrammierung viel Aufmerksamkeit geschenkt, da eine bessere **Vorsortierung** der Zugmöglichkeiten (s. 2.2.4. bis 2.2.7.) tatsächlich zu enormen Geschwindigkeitsverbesserungen führt. Man verfolgt diesen Ansatz in der Praxis meist derart, daß Informationen über Züge gespeichert werden, die ein hohes Ergebnis erzeugt haben. Sollten innerhalb der Suche Stellungen erreicht werden, in denen diese Züge erneut möglich sind, so

werden sie in der Reihenfolge der Knotenexpansionen nach vorne sortiert, da vermutet werden kann, daß sie wieder zu hohen Ergebnissen führen werden. Ist die Annahme zutreffend, so führt dies zu mehr Schnitten im Suchbaum sowie weniger Wiederholungssuchen, also zu einer Steigerung der Effizienz und somit der Geschwindigkeit der Suche.

2.1.6. Zugerzeugung und Legalität

Ein wichtiger Aspekt in der Anwendung von MinMax-Suchverfahren in der Schachprogrammierung ist die Ausführung legaler Züge. Ein Zug wird dann als legal definiert, wenn die dadurch erreichte Stellung legal ist, und eine Figur regelkonform bewegt wurde. Eine Stellung gilt dann als legal, wenn der aktive Spieler nach der Ausführung seines Zuges nicht im Schach steht; es ist also nicht möglich, sich selbst ins Schach zu stellen. Sollte ein Spieler bereits im Schach stehen und keinen legalen Zug zur Verfügung haben, so ist er Schachmatt, da es keine Zugmöglichkeit gibt, die das Schach aufheben würde (s. Anhang D „FIDE-Schachregeln“). Stehen keine legalen Züge zur Verfügung und der aktive Spieler befindet sich nicht im Schach, so handelt es sich nach den Schachregeln um ein Patt, also ein Unentschieden. Für die hier dargestellten Situationen darf in der MinMax-Suche kein Zug ausgeführt werden[6], um Schachmatt und Pattergebnisse erkennen zu können. Die Überprüfung der Legalität möglicher Züge ist also ein Aufwand, der nicht zu vermeiden ist: Es dürfen **ausschließlich** legale Züge ausgespielt werden, um korrekte Ergebnisse, im Sinne der Einhaltung der oben genannten Schachregeln, garantieren zu können.
Da der Aufwand der Legalitäts-Prüfung für jeden ausgeführten Zug geleistet werden muss, führt er zu einer deutlichen Verlangsamung der Suche. Eine Möglichkeit, diesen zusätzlichen Aufwand in Grenzen zu halten, ist die Erzeugung einer **pseudolegalen** Zugliste. Diese Zugliste enthält alle Zugvarianten, also auch Züge, die illegal sind. Die Züge werden also nicht bei ihrer Erzeugung auf Legalität geprüft, diese Prüfung findet erst bei der tatsächlichen Ausführung eines Zuges statt (s. 4.4.3.). Dies ist deswegen vorteilhaft, da die Prüfung der Legalität nicht notwendigerweise für alle Züge durchgeführt werden muß: Sollte die Suche aufgrund eines Beta-Schnitts abgebrochen werden können, so wurden nur die bis zu diesem Zeitpunkt ausgeführten Züge geprüft. Dies ist im Sinne der Legalität vollkommen ausreichend, da mögliche illegale Zugmöglichkeiten nicht ausgespielt wurden, also keinen Einfluß auf das Ergebnis haben können. Es ist also unnötig, die Legalität aller möglichen Züge a-priori zu überprüfen, dieser Aufwand kann durch sukzessive Prüfung zur Zeit der Ausführung der Züge deutlich gemindert werden.
Die Prüfung der Legalität eines Zuges kann auf zwei Arten geschehen: Wenn das Feld, von dem eine Figur wegbewegt wurde, nicht auf einer Diagonalen oder Orthogonalen des eigenen Königs liegt, so kann der Zug nicht zu einer illegalen Stellung führen, da es in diesem Fall nicht möglich ist, den eigenen König ins Schach zu stellen, der Zug ist also in jedem Fall legal. Die Überprüfung des Startfeldes läßt sich über die Abfrage einer einfachen Datenstruktur implementieren, und ist dadurch deutlich effizienter, als eine Stellung vollständig auf Legalität zu prüfen. Dieser Ansatz ist aber nur dann anwendbar, wenn der Zug nicht den eigenen König bewegt hat und kein En-Passant-Zug (s. Anhang D) vorliegt. Sind diese Bedingungen nicht gegeben, so ist man gezwungen, die Legalität eines Zuges anhand der entstehenden Stellung zu überprüfen. Hierzu müssen, nach Ausführung des Zuges, alle möglichen Züge getestet werden, die den König des aktiven Spielers angreifen könnten. Ist einer dieser Züge für den Gegenspieler möglich, so war der vorher ausgeführte Zug illegal, da der eigene König nach dessen Ausführung im Schach stehen würde, die entstehende Stellung darf nicht erzeugt werden.
In Nemesis 1.0 wird eine weitere Vereinfachung der Legalitätsprüfung angewendet, die jedoch implementationsspezifisch ist: Wenn ein bester Zug aus der Transposition Table (s. 2.1.8.) bekannt ist, so muß er nicht auf Legalität geprüft werden, da dies bereits vor dem Speichern geschehen ist.

6 Der König kann also nie geschlagen werden

2.1.7. Erkennung von Zyklen im Suchgraphen

Eine Problematik der MinMax-Suche für Schachprogramme ist die Erkennung von Zyklen im Suchablauf. Es handelt sich für ein Schachspiel nämlich tatsächlich nicht um einen Suchbaum, sondern um einen Suchgraphen[7]. So erzeugt z.B. die Zugfolge 1. b1c3 g8f6 2. c3b1 f6g8 eine Wiederholung der Ausgangsposition, also einen Zyklus im Pfad der Suche. Zyklen können sich aber auch über sehr viel mehr Züge erstrecken, sie sind nur dann ausgeschlossen, wenn innerhalb der letzten beiden Züge ein Bauer bewegt oder eine Figur geschlagen wurde, da diese Züge nicht umkehrbar sind.
Die MinMax-Suche ist von der Konzeption her nicht gut geeignet, um mit solchen Fällen umzugehen, da Zyklen im Suchablauf algorithmisch **nicht** erfasst werden. Es ist aber aus mehreren Gründen notwendig, sie zu erkennen:
Zum einen sollte man Zyklen aus Performanzgründen unbedingt vermeiden. Es ist nicht effizient, die gleiche Stellung mehrmals zu untersuchen, nur weil sie innerhalb eines Pfades mehrfach erzeugt werden kann. Eine einfache Wiederholung ist schlichtweg Zeitverschwendung, im schlimmsten Fall aber werden sich manche Pfade der Suche nur noch in Zyklen bewegen, also trotz steigender Suchtiefe keinen sinnvollen Beitrag mehr zum Ergebnis der Suche leisten können. Diese Fälle gilt es auch zur Einhaltung korrekter Ergebnisse zu verhindern.
Zum anderen gibt es im Schach die Regel des Patt durch Wiederholung. So wird eine Stellung dann als Patt gewertet, sobald sie im Laufe eines Spiels zum dritten Mal erzeugt wurde (s. Anhang D). Dieser Fall muß natürlich ebenfalls als Zyklus berücksichtigt werden, da ein Schachprogramm diese Regel kennen muß, um nicht in aussichtsreichen Situationen (mit möglicherweise sehr guten Suchergebnissen) durch eine dreifache Wiederholung ein Unentschieden zu verursachen.
Falls im Ablauf der Suche ein Zyklus erkannt wird, so muß die Suche an dieser Stelle abgebrochen werden, als Ergebnis wird ein Unentschieden (s. 2.1.10.) zurückgegeben. So kann man einerseits den überflüssigen Aufwand der Wiederholung schon besuchter Stellungen vermeiden, und außerdem die „Patt durch Wiederholung“-Regel implementieren: Man bewertet hierbei bereits die zweite Wiederholung, also den Zyklus, als Unentschieden. Eigentlich wäre erst die dritte Wiederholung ein Patt, aber auf diese Weise ist sichergestellt, daß der Computer nur dann eine Wiederholung erzeugt, wenn ein Unentschieden für ihn tatsächlich das bestmögliche Ergebnis ist, er also ein Patt durch Wiederholung nur dann anstrebt, wenn es eine Verbesserung gegenüber den anderweitig möglichen Ergebnissen darstellt. Die Erkennung von Zyklen im Suchgraphen wird meist mit Hilfe eines Hashverfahrens realisiert, worauf ich in 2.2.3. weiter eingehen werde.

2.1.8. Speichergetriebene Suche durch Zugumstellungstabellen

Generell profitieren alle Suchverfahren sehr von der Verwendung von Speicher, in dem Ergebnisse vorheriger Suchen abgelegt werden können. Die meisten MinMax-Suchalgorithmen (bis auf SSS und DUAL) sind jedoch algorithmisch darauf ausgelegt, ohne zusätzlichen Speicher zu arbeiten [Plaat; Schaeffer; Pijls; de Bruin 95]. Da es im Schach aber möglich ist, durch **Zugumstellung** (Änderung der Reihenfolge gespielter Züge) die gleiche Stellung auf verschiedenen Pfaden zu erreichen, wäre es nur logisch, Informationen der Suche zu speichern, um für diesen Fall die Suche mit weniger Aufwand terminieren zu können. So erzeugen z.B. die beiden Eröffnungsvarianten 1. e2e4 g8f6 2. b1c3 ... und 1. b1c3 g8f6 2. e2e4 ... die gleiche Stellung[8]. Sollten beim ersten Erreichen der Stellung bereits ausreichend Informationen gespeichert worden sein, so kann man diese bei einem erneuten Erreichen zurückgeben, anstatt die Stellung nochmal zu untersuchen. Um so überflüssige Wiederholungen der gleichen Suche zu vermeiden, werden Zugumstellungstabellen,

7 Ich werde jedoch aus Gründen der Anschaulichkeit bei allgemeinen Beschreibungen weiter den Ausdruck Suchbaum verwenden

8 Im Unterschied zu 2.1.7 liegen sie aber in unterschiedlichen Suchpfaden

im Englischen **Transposition Tables**, verwendet. Sie führen immer dann zu einer Steigerung der Effizienz, wenn im Ablauf einer MinMax-Suche eine Stellung mehrfach erreicht wird. Dies kann auch durch Wiederholungssuchen verursacht werden, die ein Bestandteil vieler Verfahren sind (s. 2.1.3., 2.2.1., 2.2.2., 2.2.6. und 2.2.8.).
In der Transposition Table speichert man die Tiefe, zu der eine Stellung analysiert wurde, sowie das höchste erreichte Ergebnis. Außerdem speichert man zu jeder analysierten Stellung den Zug, der dieses Ergebnis geliefert hat. Dieser Zug wird als erster untersucht, falls die Stellung erneut erreicht wird, um die Vorsortierung der Knoten zu verbessern. Es wird außerdem gespeichert, ob es sich bei dem Suchergebnis um einen exakten Wert, eine Obergrenze (also das Ergebnis eines Alpha Fails) oder eine Untergrenze (das Ergebnis eines Beta Cutoffs) handelt. Diese Daten können bei einer wiederholten Untersuchung der Stellung verwendet werden, um einen Alpha- oder Beta-Schnitt durchzuführen, falls ein hierzu ausreichendes Ergebnis vorhanden ist. Sollte mit dem Ergebnis aus der Transposition Table nicht abgeschnitten werden können, so kann man immer noch versuchen, das aktuelle Suchfenster zu verkleinern, da möglicherweise eine engere Ober- oder Untergrenze aus einer früheren Suche bekannt ist. Dies würde zu mehr Alpha/Beta - Schnitten in den Unterbäumen des aktuellen Knotens führen.
Wie eine Transposition Table in der Schachprogrammierung implementiert wird, werde ich in 2.2.3. erläutern. Der praktische Ablauf der Speicher- und Lesevorgänge sowie die Verarbeitung der gewonnenen Daten ist in 4.4.3. dargestellt. Die technischen Details der Implementation einer Transposition Table in Nemesis sind 4.5.3. zu entnehmen.

2.1.9. Selektive Suche durch Search Extensions

Bereits 1949 hat Claude Shannon gezeigt, daß es zwei verschiedene Ansätze gibt, die MinMax-Suche in einer Schachanwendung durchzuführen [Shannon 49]. Der Ansatz, der in der modernen Schachprogrammierung verwendet wird, ist im Englischen als brute force - Methode bekannt. Dies bedeutet, daß der Spielbaum vollständig (also mit „roher Gewalt“) untersucht wird, wobei **alle Zugkombinationen** berücksichtigt werden, also auch solche, die einem menschlichen Spieler offensichtlich als unsinnig erscheinen würden. Shannon hatte 1949 noch bezweifelt, daß dieser Ansatz zur Realisierung eines Schachcomputers tauglich sei, da er sowohl der menschlichen Intuition als auch der menschlichen Spielweise widerspricht; außerdem bemängelte er den rechnerischen Aufwand dieses Verfahrens. Deswegen hat er einen weiteren Ansatz entwickelt, den sog. Shannon-Typ-B: Dieser Ansatz geht davon aus, daß es sinnvoller ist, nicht ungesteuert alle Zugvariationen zu erzeugen, sondern stattdessen eine Selektion durchzuführen, d.h. nur eine **Untermenge** der möglichen Züge zu untersuchen. In den 60er und 70er Jahren wurde dieser Ansatz von den meisten Schachprogrammen verwendet, da er zur Realisierung nur einen Bruchteil der Ressourcen benötigt. Sollten in einer MinMax-Suche statt der durchschnittlich etwa 35 Zugmöglichkeiten nur 15 Züge erzeugt werden, so reduziert sich der Aufwand einer Suche zur Tiefe 6 auf nur 0,6% des ursprünglichen Aufwands.
Diese Technik bringt also enorme Kostenersparnisse, sie hat allerdings auch einen gravierenden Nachteil: Nach welchen Kriterien sind Züge auszuwählen, die untersucht werden sollen? Was im Umkehrschluß bedeutet: Nach welchen Kriterien bestimmt man Züge, die von der weiteren Suche ausgeschlossen werden können? Zur Lösung dieses Problems sind ganz verschiedene Ansätze entwickelt worden, die jedoch alle mit deutlichen Mängeln behaftet sind, da es kaum möglich ist, die notwendige Zugauswahl so zu optimieren, daß keine strategischen Schwächen entstehen. Es gibt im Schach nämlich oft Situationen, in denen ein bestimmter Zug nicht bedeutsam erscheint, aber in der Kombination mit einer späteren Entwicklung auf einmal Signifikanz entwickelt. Diese Züge zu finden ist das eigentliche Problem der **selektiven Suche**, da es kaum eindeutige Kriterien gibt, an denen man den strategischen Wert eines Zuges a-priori bestimmen könnte. Dieser strategische Wert ergibt sich tatsächlich erst durch eine brute force MinMax-Suche, da nur hier alle

möglichen weiteren Kombinationen erzeugt werden, die ihn erst bestimmen können. Es hat sich dann auch im Laufe der 70er Jahre gezeigt, daß Programme mit einer selektiven Suche, auf Grund der oben dargestellten Schwäche, unterlegen sind; es haben sich die Programme durchgesetzt, die nach dem Ansatz Shannon-Typ-A vorgehen (s. 3.1.), der auch heute noch verwendet wird.
Der Gedanke eines selektiven Elements in der MinMax-Suche wurde später aber wieder aufgegriffen, und zwar in Form der sog. Search Extensions. Damit ist gemeint, die Tiefe der Suche an bestimmten Knoten zu erhöhen, wenn diese gewissen Kriterien entsprechen; die Suche wird also dadurch selektiv, daß einzelne Pfade im Suchbaum tiefer untersucht werden als andere. Search Extensions sollten immer dann ausgelöst werden, wenn es wahrscheinlich scheint, daß in der Folge des aktuelle Knotens wichtige Veränderungen der Spielsituation entstehen, um eben diese Veränderungen genauer erkennen zu können. Dies führt zu einer signifikanten Steigerung der Qualität der gefundenen Ergebnisse; das Verfahren führt außerdem nicht zu der oben gezeigten Schwäche, da ja nach wie vor alle möglichen Zugkombinationen erzeugt werden.
Durch Anwendung von Search Extensions können jedoch **Inkonsistenzen** in der Suche entstehen, da Suchpfade, in denen Extensions ausgelöst wurden, Informationen aus größerer Tiefe enthalten, und dadurch in der Verrechnung mit anderen Suchpfaden eine Verfälschung der Ergebnisse erzeugen können. Aus diesem Grund werden Search Extensions in ihrer Tiefe limitiert, um zu verhindern, daß der Unterschied zweier benachbarter Pfade in der Suchtiefe zu groß werden kann.
Die Implementation von Search Extensions in Nemesis sowie die dabei verwendeten Kriterien zu ihrer Auslösung werde ich in Kapitel 4.5.4. vorstellen.

2.1.10. Bewertung von Knoten

Wie bereits in 2.1.1. dargestellt wurde, gibt es verschiedene Situationen, in denen es notwendig ist, Knoten zu bewerten. Diese Situationen unterscheiden sich in zwei Kategorien: Zum einen ist eine Bewertung immer dann notwendig, wenn ein Blatt erreicht wird, also ein Knoten, der ein definiertes **Spielende** darstellt. Hier gibt es wiederum verschiedene Möglichkeiten: Falls ein Spieler Schachmatt ist, so hat er das Spiel verloren. Das Blatt muß in dieser Situation einen sehr hohen negativen Wert zurückgeben, der für einen maximierenden Spieler das schlechtestmögliche Ergebnis darstellt. In einer praktischen Anwendung nimmt man hierfür einen Wert, der weit unter den sonst möglichen Bewertungen liegt. Ein weiteres definiertes Spielende ist ein Patt. Ein Patt kann auf verschiedene Weise erzeugt werden, es bedeutet nach den Schachregeln aber immer ein Unentschieden. Die einfachste Art, dieses Ergebnis zu repräsentieren, ist der Wert null, da er für ein Nullsummenspiel tatsächlich ein Unentschieden bedeutet. Es gibt auch die Möglichkeit einen sog. contempt factor zurückzugeben, der angibt, ob man mit einem Unentschieden zufrieden ist. Ist der contempt factor z.B. minus dem Wert eines Springers, so wird das Programm ein Patt nur dann anstreben, wenn alle anderen möglichen Ergebnisse noch schlechter als der Verlust eines Springers sind. Daher kommt auch der Name contempt factor, den man in etwa als „Geringschätzungsfaktor“ übersetzen könnte. Hiermit ist gemeint, daß ein Programm mit einem niedrigen contempt-factor den Gegner als schlecht einschätzt, weil es ein Unentschieden erst dann für akzeptabel hält, wenn die anderen möglichen Ergebnisse noch schlechter sind als eben dieser contempt factor. Dieser Wert kann willkürlich gesetzt werden, es gibt aber auch die Möglichkeit, ihn aus Angaben zur Spielstärke des Gegenspielers zu errechnen.
Die andere Situation, in der es notwendig ist, Knoten zu bewerten, ist die Überschreitung des Suchhorizonts. Dieser Fall ist deutlich schwieriger zu handhaben, da die Knoten hier kein definiertes Spielende darstellen. Die Bewertung dieser Stellungen kann also allenfalls eine Schätzung sein; die **Bewertungsfunktion**, die diesen Wert errechnet, ist demzufolge eine heuristische Funktion. Obwohl es sehr viele verschiedene Kriterien gibt, an denen man die Bewertung einer Stellung bestimmen kann, wird immer der aktuelle **Materialwert** zugrunde gelegt. Jede Figur hat hierfür einen spezifischen Wert, ihren Materialwert. Der Materialwert einer Stellung

ergibt sich aus der Summe der Materialwerte der vorhandenen Figuren, wobei weiße Figuren positiv und schwarze Figuren negativ bewertet werden, in einer Stellung mit ausgeglichenen Figurenverhältnissen ist der Materialwert also null. Zu diesem Wert werden nun verschiedene **positionelle Faktoren** addiert: Hierzu gehören Faktoren, die die Positionierung der eigenen Figuren auf dem Spielfeld bewerten, ihre Verhältnisse untereinander sowie zu den gegnerischen Figuren. Außerdem werden Faktoren wie das Recht, eine Rochade auszuführen (s. Anhang D), die Deckung des eigenen Königs, Angriffe auf den gegnerischen König etc... berücksichtigt.
Es ist nicht einfach, diese Bewertungsfunktion zu optimieren, da alle Kriterien auf ihre **Eindeutigkeit** geprüft werden müssen. Es galt z.B. lange Zeit die Annahme, man sollte einen eigenen Turm auf der Grundreihe der gegnerischen Bauern positiv bewerten, da er die gegnerische Stellung unter Druck setzt. Dies ist auch meistens der Fall, es gibt jedoch auch Situationen, z.B. in Endspielen mit wenigen Figuren, wo die Tatsache, daß ein Turm auf dieser Reihe steht, keine Bedeutung mehr hat. In diesen Fällen führt dieses Kriterium also zu einer Verschlechterung der Genauigkeit der Bewertung; man muß sich in diesem Zusammenhang immer vor Augen halten, daß die Bewertungsfunktion auf alle theoretisch möglichen Stellungen anwendbar sein muß, da man nicht weiß, welche davon tatsächlich im Laufe eines Spiels generiert werden.
Ein weiterer Punkt, der zu beachten ist, ist die **Gewichtung der Kriterien** untereinander. So ist es z.B. nicht sinnvoll, einen Bauernvorstoß positiv zu bewerten, wenn dieser Bauer gleichzeitig ein Teil der eigenen Königsdeckung war, und sein Vorstoß diese verschlechtert hat. Das Kriterium der Königsdeckung sollte ein höheres Gewicht haben als die Bewertung des Bauernvorstoßes, um die unterschiedliche Signifikanz der Faktoren zu repräsentieren. Es gibt viele Faktoren, die auf solche Weise miteinander interagieren, mögliche Ergebnisse sind deshalb oft schwer abzuschätzen.
Um eine gute Bewertung zu erreichen, ist es notwendig, möglichst eindeutige Kriterien auszuwählen und diese sehr genau gegeneinander zu gewichten. Wie ich diese Anforderungen an die Bewertungsfunktion berücksichtigt habe, werde ich in 4.5.5. ausführlich darstellen.

2.1.11. Horizont-Effekt und Schlagfallanalyse

Die größte Problematik in der Anwendung der MinMax-Suche ist der sog. Horizont-Effekt. Nehmen wir folgende Situation an: Der Materialwert der aktuellen Stellung ist ausgeglichen, aber der aktive Spieler kann den Verlust einer wichtigen Figur in den folgenden Zügen nicht mehr verhindern. Im Ablauf der MinMax-Suche wird dieses Problem erkannt, gleichzeitig findet die Suche jedoch eine scheinbare Lösung: Wenn genügend andere unwichtige Züge vorher gespielt werden, so gelingt es, den Verlust der Figur hinter den Suchhorizont zu **verschieben**. Die Suche wird nur bis zu einer begrenzten Tiefe betrieben, an der weitere Unterbäume einfach abgeschnitten werden. Gelingt es also genügend Züge zu finden, die den Verlust der Figur herausschieben, so wird die Suche diese Züge besser bewerten, da der Verlust hinter den Horizont, also außerhalb der Menge der untersuchten Knoten, verschoben wird. Der Materialverlust wird auch im Ergebnis der Suche nicht erkannt. Im schlimmsten Falle wird die Suche sogar Ergebnisse bevorzugen, die eine leichtere (weniger wichtige) Figur opfern, um den größeren Verlust zu verschieben. Da dieser Verlust später aber doch eintritt, bringt diese Spielweise ein noch schlechteres Ergebnis, als durch den Verlust der einen Figur unvermeidbar gewesen wäre. Durch die dargestellte Verschiebung entstehen Inkonsistenzen in der Suche und dadurch unrealistische Ergebnisse, dieses Verhalten nennt man Horizont-Effekt.
Etwas abstrakter betrachtet könnte man das Problem so formulieren, daß es gilt, nur Stellungen als Blätter, also als heuristisch zu bewertende Positionen, zu akzeptieren, die „ruhig“ sind. Ruhig wird hier definiert als Abwesenheit möglicher Schlagkombinationen, die zu einem größeren Wechsel der Bewertung führen könnten. Diese Problematik wurde bereits frühzeitig erkannt und beschrieben, z.B. von [Shannon 49 / Turing et al 53]. Um sie zu beherrschen, gibt es verschiedene Möglichkeiten. Eine davon wäre die Durchführung einer Static Exchange Evaluation (s. 2.2.5.) an

allen potentiellen Blättern. Da SEE jedoch keine Ergebnisse hoher Genauigkeit liefert, wird dieses Verfahren im Normalfall hier nicht eingesetzt. Stattdessen ist es notwendig, am Suchhorizont eine weitere Suche durchzuführen; dies geschieht in Form einer Schlagfallanalyse, die man im Englischen **Quiescence Search** nennt, was man als „Suche nach Ruhe“ übersetzen könnte. Diese Suche wird analog zur normalen Suche ausgeführt, der größte Unterschied ist die Begrenzung der möglichen Züge auf **Schlagzüge**. Sollte ein Spieler im Schach stehen, so werden alle Zugmöglichkeiten erzeugt, um Schachmatt korrekt erkennen zu können: In der Quiescence Search werden also alle möglichen Schlagkombinationen sowie die daraus entstehenden Mattangriffe ausgespielt. Die Tiefe dieser Suche wird durch eine mögliche Maximaltiefe sowie die natürliche Begrenzung der Menge möglicher Schlagfolgen eingeschränkt. Sollte in der Quiescence Search eine Stellung erreicht werden, in der es keine Schlagzüge mehr gibt, so wird die Bewertungsfunktion aufgerufen, da diese Stellung als ruhig bezeichnet werden muß.
Das Verfahren der Quiescence Search soll den oben gezeigten Effekt der Verschiebung hinter den Horizont verhindern, da ja jetzt alle Schlagzüge ausgespielt werden **müssen**, also eben nicht mehr verschoben werden können. Es ist jedoch auch zu beachten, daß es keinen „Schlagzwang“ geben darf: Jeder Spieler muß die Möglichkeit haben, den Wert einer Schlagabfolge nicht anzunehmen, wenn er sich durch einen Schlagzug gegenüber der lokalen Bewertung am aktuellen Knoten verschlechtert. Es liegt hier die Annahme zugrunde, daß es in jeder Stellung mindestens eine Zugmöglichkeit gibt[9], die das statische Ergebnis des Knotens hält, man geht also von einer Stabilität des lokalen Ergebnisses aus. Eine Vereinfachung der Quiescence Search, die allgemein Anwendung findet, basiert auf der oben genannten Annahme: Der sog. „Stand-Pat“ - Beta-Schnitt versucht, mit dem Ergebnis der lokalen Evaluation einen Beta-Schnitt zu verursachen, um auf diese Weise die Quiescence Search früher abbrechen zu können[10]. Er kann nicht angewendet werden, wenn der aktive Spieler im Schach steht (s.o.).
Die obengenannten Vereinfachungen führen in der Quiescence Search zu einer Verringerung der Genauigkeit der Suchergebnisse; die Qualität sinkt, da die Annahme der Stabilität lokaler Ergebnisse möglicherweise fehlerhafte Einschätzungen liefert. Da diese Ergebnisse jedoch in größtmöglicher Entfernung der Wurzel erzeugt werden, wird dieser Effekt teilweise durch die aus der Verrechnung ihrer enormen Anzahl entstehende Glättung kompensiert.
Die Schlagfallanalyse erzeugt großen Overhead, da sie für alle Knoten am Suchhorizont ausgeführt werden muß. Hierbei muß bedacht werden, daß aufgrund des exponentiellen Wachstums pro Suchtiefe und der Anwendung verschiedener Schnittverfahren innerhalb des Baums in einer typischen Suche etwa die Hälfte aller expandierten Knoten am Suchhorizont liegen! Da für alle diese Knoten eine Quiescence Search ausgeführt werden muß, ist der zusätzliche Aufwand enorm, er muß aber unbedingt geleistet werden, um den dargestellten Horizont-Effekt zu verhindern: Ohne Quiescence Search können wesentlich höhere Suchtiefen erreicht werden, die Suchergebnisse sind aber derart fehlerbehaftet, daß sie für eine ernsthafte Anwendung nicht tauglich sind. Die Quiescence Search liefert zwar auch keine endgültigen Ergebnisse, sie ist jedoch der bestmögliche Kompromiß zwischen zusätzlichem Aufwand und erreichbarer Genauigkeit.

9 In der Menge aller möglichen Züge, nicht nur der Schlagzüge

10 Der Name „Stand-Pat“ ist aus dem Pokerspiel entlehnt, wo er bedeutet, das aktuelle Ergebnis als ausreichend gut anzunehmen und aufgrund dieser Annahme keine weiteren Aktionen durchzuführen

2.2. Verfahren der Schachprogrammierung

2.2.1. Der PVS/NegaScout – Algorithmus

Der PVS-Algorithmus wurde von Marsland und Campbell 1982 eingeführt [Marsland; Campbell 82] und hat seitdem bewiesen, eine Verbesserung gegenüber einfachen Alpha/Beta-Verfahren darzustellen [Marsland 83]. Er wurde von Alexander Reinefeld in der Notation etwas vereinfacht und um eine kleine Verbesserung erweitert, diese Variante heißt NegaScout [Reinefeld 83].
Die Idee des PVS/NegaScout-Algorithmus besteht darin, nach der Bewertung des ersten Unterbaums, die analog zum Alpha/Beta-Algorithmus erfolgt, alle weiteren Unterbäume a-priori als **unterlegen** zu betrachten und **Nullfenstersuchen** durchzuführen, um diese Unterlegenheit in Form von Obergrenzen zu beweisen.
In einer praktischen Anwendung ermittelt man hierfür den ersten Blattwert als MinMax-Wert *n*. Dann versucht man, mit Hilfe von Nullfenstersuchen mit dem Suchintervall [*n, n* + 1] zu beweisen, daß die Alternativen den MinMax-Wert nicht beeinflussen können, da ihre Ergebnisse die Obergrenze des Nullfensters nicht erreichen: Die Nullfenstersuchen sind dann erfolgreich, wenn die restlichen maximierenden Nachfolger kleiner gleich *n* und die restlichen minimierenden Nachfolger größer gleich *n* sind. Gelingt der Beweis, so hat man mit einem minimalem Aufwand an Knotenexpansionen gezeigt, daß *n* der tatsächliche MinMax-Wert ist. Schlägt der Beweis hingegen fehl, muß der betreffende Unterbaum im Rahmen einer **Wiederholungssuche** mit geöffnetem Suchfenster noch einmal durchsucht werden. Sein MinMax-Wert wird dann als neuer Referenzwert *n* für weitere Nullfenstersuchen verwendet.
Die Vorgehensweise des PVS/NegaScout - Algorithmus ist vor allem dann effektiv, wenn mit geeigneten Heuristiken sichergestellt wird, daß aussichtsreiche Züge zuerst untersucht werden (s. 2.1.5.); das Nullfenster also auf eine für die weiteren Knoten realistische Obergrenze angehoben wurde. Die Ersparnisse der Suche mit einem Nullfenster sind dann so groß, daß der Mehraufwand gelegentlicher Wiederholungssuchen bei weitem aufgewogen wird.

Darstellen möchte ich hier die NegaScout-Variante, da sie gegenüber PVS etwas kompakter ist:

```
int NegaScout(Stellung s, int tiefe, int alpha, int beta) {

   if (tiefe == 0 OR Blatt(s)) RETURN Bewertung(s);

   int a = alpha;
   int b = beta;

   Finde Nachfolger s_1,...,s_w von s;

   for ( int i = 1; i <= w; i++) {

      int temp = -NegaScout(s_i, tiefe-1, -b, -a);

                                                    //Wiederholungssuche?
      if (temp > a) AND (temp < beta) AND (i > 1) AND (tiefe > 1)
         a = -NegaScout(s_i, tiefe-1, -beta, -temp);

      a = MAX(a, temp);                             //Alpha – Anpassung

      if (a >= beta) return (a);                    //Beta - Schnitt

      b = a + 1;                                    //Neues Nullfenster
   }

   return (a);
}
```

Ein wesentlicher Unterschied zur Alpha/Beta – Suche ist das Vorhandensein der Wiederholungssuche. Sie wird immer dann durchgeführt, wenn der untersuchte Zug nicht der erste war (also überhaupt mit einem Nullfenster gesucht wurde), das Ergebnis der Suche besser als der Alpha-Wert ist (die Unterlegenheit also nicht bewiesen werden konnte), aber nicht größer gleich dem Beta-Wert (sonst fällt diese Variante sowieso einem Beta-Schnitt zum Opfer). Außerdem müssen noch mindestens zwei Halbzüge bis zum Erreichen der maximalen Tiefe verbleiben, dies ist Alexander Reinefelds Verbesserung, das sog. „fail-soft refinement“ [Reinefeld 83] . Sollte diese Bedingung nicht erfüllt sein, so läßt sich beweisen, daß das Ergebnis der Nullfenstersuche dem tatsächlichen Ergebniswert entspricht. Für eine Wiederholungssuche wird das Ergebnis der Nullfenstersuche als neuer Alpha-Wert verwendet: Die Nullfenstersuche hat eine neue Untergrenze auf das Ergebnis bewiesen, und diese Untergrenze kann nun benutzt werden, um möglichst viele Schnitte in der Wiederholungssuche zu erzeugen.

Auf den ersten Blick erscheint die Nullfenstersuche als ein riskantes Unterfangen, da Wiederholungssuchen nie gänzlich vermieden werden können. Empirische und analytische Untersuchungen [Reinefeld 89] beweisen jedoch, daß die mit dem Nullfenster erzielten Einsparungen den Mehraufwand gelegentlicher Wiederholungssuchen bei weitem übertreffen. Die Stärke des PVS/NegaScout – Algorithmus beruht darauf, daß es effizienter ist, die Unterlegenheit eines Unterbaums mit einer Nullfenstersuche zu beweisen, als seinen exakten MinMax-Wert zu berechnen, was sich im Nachhinein oft als überflüssig herausstellt. Da der PVS/NegaScout – Algorithmus zusätzlich zu den Schnitten der Nullfenstersuche auch die regulären Alpha- und Beta-Schnitte realisiert, gilt die folgende Dominanzrelation: **Jeder Knoten, der von PVS/NegaScout expandiert wird, muß auch von Alpha/Beta expandiert werden, umgekehrt jedoch nicht** [Reinefeld 89]. Um die Menge der möglichen Schnitte noch zu erhöhen, kann auch der PVS/NegaScout – Algorithmus mit einem Aspiration Window verwendet werden.

Der dargestellte Baum veranschaulicht den Ablauf des Algorithmus:

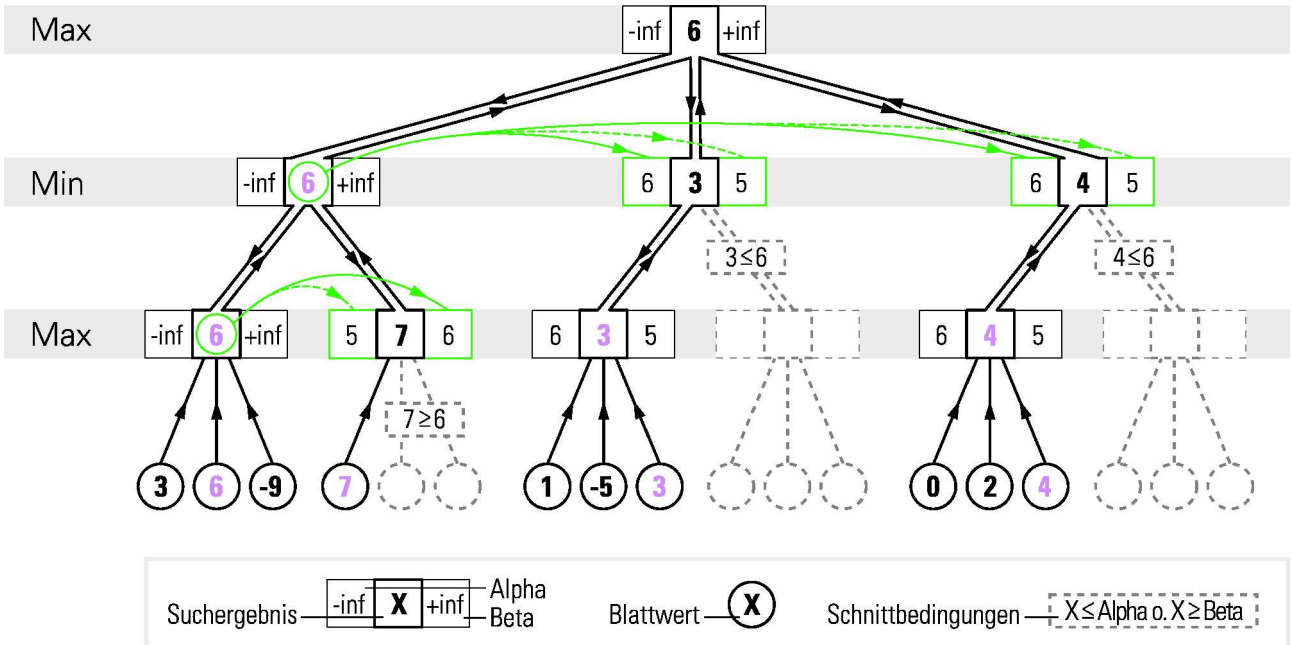


Abb. 2.5: Ablauf einer PVS/NegaScout – Suche

Für den mittleren und den rechten Unterbaum der Wurzel ist bekannt, daß der maximierende Spieler mindestens das Ergebnis 6 erreichen kann. Es wird nun versucht, mit Hilfe eines

Nullfensters zu beweisen, daß diese Unterbäume kein besseres Ergebnis liefern können, was zu den dargestellten Alpha-Schnitten führt. Die abgeschnittenen Unterbäume werden nicht weiter untersucht, da ihre Teilergebnisse bereits die Untergrenze des Nullfensters verletzen. Die Nullfenstersuchen haben beweisen können, daß diese Unterbäume Obergrenzen mit den Werten 3 bzw. 4 besitzen. Da der maximierende Spieler ein höheres Ergebnis aus dem linken Unterbaum sichergestellt hat, können sie vernachlässigt werden. Die Annahme der Unterlegenheit dieser Unterbäume war also richtig, die Anwendung einer Nullfenstersuche hat den Ergebniswert deshalb mit einer geringeren Zahl an expandierten Knoten (gegenüber einer einfachen Alpha/Beta-Suche) beweisen können.

2.2.2. Der MTD(f)-Algorithmus

MTD(f) ist ein relativ neuer MinMax-Suchalgorithmus, der in der Praxis bewiesen hat, effizienter als PVS/NegaScout sein zu können [Plaat; Schaeffer; Pijls; de Bruin 94]. Der Name des Algorithmus ist eine Kurzform für Memory Test Driver(n, f), was in etwa „Speichergetriebener Test mit Knoten n und Wert f“ bedeutet.
MTD(f) gewinnt seine Effizienz dadurch, daß ausschließlich Nullfenstersuchen durchgeführt werden. Dies erleichtert die Traversierung des Spielbaums gegenüber Verfahren wie Alpha/Beta-Suche und PVS/NegaScout, die mit größeren Suchfenstern beginnen. Nullfenstersuchen führen zu wesentlich mehr Schnitten im Suchbaum, sind im Ablauf also performanter.
Der Algorithmus verwendet einen **mehrfachen** Aufruf einer **Nullfenstersuche** zur gleichen Tiefe, um eine Annäherung an den MinMax-Wert zu ermöglichen. Jeder Aufruf gibt dabei eine neue Grenze auf den gesuchten Wert zurück (s. 2.1.4.), diese Werte werden in *untereGrenze* und *obereGrenze* gespeichert und bilden so ein Intervall um den gesuchten MinMax-Wert. Anfänglich reicht dieses Intervall von plus bis minus unendlich, nähert sich jedoch schnell an den tatsächlichen Wert an. Wenn die obere und die untere Grenze kollidieren, ist der MinMax-Wert gefunden.
MTD(*f*) muß ein Suchverfahren verwenden, das den gefundenen Wert zu einem Knoten speichern kann und diesen in einer Wiederholungssuche aus dem Speicher liest, um eine weitere Suche des gleichen Baums zu vereinfachen (s. 2.1.8.). Dies ist grundlegend für den Performanzgewinn gegenüber dem PVS/NegaScout – Algorithmus (ebenfalls mit Speicherunterstützung), da ansonsten die hohe Anzahl der Wiederholungssuchen einen zu großen Aufwand erzeugen würde [Plaat; Schaeffer; Pijls; de Bruin 95].

```
int MTDF (Stellung s, int tiefe, int nullfenster) {

  int wert = nullfenster;
  int beta;

  int unterGrenze = -INFINITY;
  int oberGrenze = +INFINITY;

  while (unterGrenze < oberGrenze) {

    if (wert == unterGrenze) beta = wert + 1;       //Neues Nullfenster
    else beta = wert;

    wert = NegaMaxAlphaBeta(s, tiefe, beta-1, beta);//Nullfenstersuche

    if (wert < beta) oberGrenze = wert;             //Unter- oder Obergrenze?
    else unterGrenze = wert;
  }

  return (wert);
}
```

Der Ablauf einer Suche sieht typischerweise so aus:

```
Start Search: Lowerbound = -1000000 Upperbound = 1000000 Nullwindow = 0
Search Result: Lowerbound = -1000000 Upperbound = -22 Nullwindow = -22
Search Result: Lowerbound = -1000000 Upperbound = -28 Nullwindow = -28
Search Result: Lowerbound = -1000000 Upperbound = -29 Nullwindow = -29
Search Result: Lowerbound = -1000000 Upperbound = -50 Nullwindow = -50
Search Result: Lowerbound = -1000000 Upperbound = -55 Nullwindow = -55
Search Result: Lowerbound = -1000000 Upperbound = -56 Nullwindow = -56
Search Result: Lowerbound = -1000000 Upperbound = -57 Nullwindow = -57
Search Result: Lowerbound = -1000000 Upperbound = -69 Nullwindow = -69
Search Result: Lowerbound = -1000000 Upperbound = -70 Nullwindow = -70
Search Result: Lowerbound = -1000000 Upperbound = -71 Nullwindow = -71
Search Result: Lowerbound = -71 Upperbound = -71 MinMax Value = -71
```

Die Suche wurde mit einem anfänglichen Nullfenster mit dem Wert null gestartet. Die erste Nullfenstersuche beweist eine Obergrenze von -20, die folgenden Suchen verschieben diese Obergrenze bis zu dem Wert -71. Dann gelingt es -71 auch als Untergrenze zu beweisen, somit ist der MinMax-Wert gefunden.
MTD(*f*) wird dadurch effizient, daß ausschließlich Nullfenstersuchen durchgeführt werden, hierfür wird der nach jedem Durchlauf auf das Ergebnis der vorherigen Nullfenstersuche angepasste Beta-Wert verwendet. Damit der MTD(f)-Algorithmus noch effizienter abläuft, ist eine initiale Annäherung an das gesuchte Ergebnis wünschenswert:

```
Start Search: Lowerbound = -1000000 Upperbound = 1000000 Nullwindow = -60
Search Result: Lowerbound = -1000000 Upperbound = -69 Nullwindow = -69
Search Result: Lowerbound = -1000000 Upperbound = -70 Nullwindow = -70
Search Result: Lowerbound = -1000000 Upperbound = -71 Nullwindow = -71
Search Result: Lowerbound = -71 Upperbound = -71 MinMax Value = -71
```

Der Algorithmus wurde hier mit einem Nullfenster gestartet, das dem Ergebnis deutlich näher liegt als im vorherigen Beispiel, es ist einleuchtend, daß dadurch der Aufwand des Verfahrens deutlich gesenkt wird. Je besser die anfängliche Annäherung ist, desto effizienter arbeitet der Algorithmus, da weniger Durchläufe, und damit weniger Wiederholungssuchen, notwendig sind, um auf den MinMax-Wert zu konvergieren. Um einen realistischen Anfangswert zu erreichen, bietet sich die Verwendung eines Iterative Deepening Framework an, auf das ich im Weiteren noch eingehen werde (2.2.6.).
Aus algorithmischen Gründen kann mindestens eine Wiederholung der Suche nicht vermieden werden: Im besten Falle würde die erste Suche eine obere Grenze auf das Ergebnis liefern und die zweite Suche diesen Wert als untere Grenze bestätigen, im Normalfall jedoch werden deutlich mehr Wiederholungen benötigt, in einer durchschnittlichen Anwendung etwa 5 bis 15. Der MTD(f) – Algorithmus profitiert stark von der Möglichkeit, Ergebnisse zu speichern, da in seinem Ablauf die Wiederholung einer Suche zur gleichen Tiefe bereits fest vorgesehen ist. Wenn man nun die Ergebnisse eines Knotens speichert, kann man bei einer Wiederholungssuche diese Ergebnisse verwenden, um die Suche mit weniger Aufwand durchführen zu können. Dies führt zu enormen Kostenersparnissen, mit deren Hilfe der MTD(f)-Algorithmus dann tatsächlich performanter als die bisher dargestellten Suchverfahren ist.

2.2.3. Hashtables und Zobrist-Schlüssel

In der Schachprogrammierung werden ausschließlich Hashtables zur Speicherung von Ergebnissen verwendet, weil sie direkten Speicherzugriff ermöglichen. Für jede rekursive Knotenexpansion muß ein Eintrag im Speicher geprüft werden, andere Verfahren sind hierzu ungeeignet, da sie zu langsam sind.

Um Hashtables zu verwenden, muß für jede Stellung ein eindeutiger **Hashwert** errechnet werden können. Hierzu verwendet man das Verfahren von Zobrist [Zobrist 70]. Dieses Verfahren verwendet für jede Figur bezüglich jeder möglichen Position auf dem Schachbrett einen unterschiedlichen Schlüssel gleicher Länge. Der Gesamtschlüssel einer Stellung wird dadurch erzeugt, daß die einzelnen Schlüssel aller vorhandenen Figuren mit der XOR-Funktion verknüpft werden. Dieses Verfahren zur Schlüsselerzeugung ist vor allem deshalb gut geeignet, da durch die Symmetrieeigenschaften der XOR-Funktion die Schlüssel auch inkrementell gehandhabt werden können: Nachdem ein Zug ausgeführt wurde, wird der Schlüssel der betreffenden Figur bezüglich ihres Startfeldes durch erneutes Anwenden der XOR-Funktion aus dem Gesamtschlüssel entfernt und ein neuer Schlüssel bezüglich des Zielfeldes hinzugefügt. Die XOR-Funktion bringt außerdem den Vorteil, daß der Schlüssel einer Position komplett verändert wird, wenn eine Figur bewegt wurde, die Schlüssel haben dadurch eine gute Verteilung, die die Wahrscheinlichkeit von Schlüsselkollisionen vermindert. Bei der Anwendung von Hashverfahren können Kollisionen nie ganz ausgeschlossen werden, in der Schachprogrammierung verwendet man deshalb üblicherweise Schlüssel mit 64-Bit Länge, viele Implementationen verwenden sogar noch einen zweiten Schlüssel, um diesen bei einer Kollision gegenprüfen zu können (s. 4.4.1. und 4.5.3.).
Bei der Erzeugung von Zobrist-Schlüsseln gilt es zu beachten, daß weitere Elemente einer Stellung ebenfalls im Schlüssel repräsentiert werden müssen: Dies gilt für das Recht eine Rochade auszuführen sowie die Möglichkeit eines En-Passant-Zuges. Außerdem wird ein eigener Schlüssel angewendet, um festzulegen, welcher Spieler gerade am Zug ist. Dies sind ebenfalls Bestandteile der aktuellen Position, sie ergeben sich jedoch nicht aus den vorhandenen Figuren, sondern vielmehr aus dem vorherigen Ablauf des Spiels. Diese Informationen müssen unbedingt in die Schlüsselerzeugung miteinbezogen werden, um Stellungen anhand ihres Schlüssels einwandfrei identifizieren zu können.
Hashtables werden in der Schachprogrammierung vor allem genutzt, um, wie in 2.1.8. erläutert, Ergebnisse der Suche speichern zu können. Die **Verwertbarkeit** eines Suchergebnisses hängt dabei von seinem Verhältnis zu den lokalen Alpha/Beta-Werten ab: Liegt das Suchergebnis eines Knotens innerhalb des lokalen Suchfensters, so handelt es sich um ein exaktes Ergebnis. Ist das Ergebnis kleiner gleich Alpha, so handelt es sich um einen Alpha Fail, ist das Ergebnis größer gleich Beta, so hat ein Beta Cutoff stattgefunden. Die Transposition Table verwendet ein Flag, um das Suchergebnis nach diesen Kriterien kategorisieren zu können. Sobald die gleiche Position in einer anderen Suche erreicht wird, können Ergebnisse, je nach ihrer durch das Flag beschriebenen Aussagekraft, wiederverwendet werden. Dies ist jedoch nur möglich, wenn die Suchtiefe des gespeicherten Ergebnisses größer oder gleich der aktuellen Suchtiefe ist, denn nur dann ist der Informationsgehalt des gespeicherten Eintrages groß genug, um keine Inkonsistenzen in der aktuellen Suche zu verursachen[11].
Hashtables werden aber auch zu anderen Zwecken verwendet. So benutzt man sie, um die Schlüssel bereits besuchter Stellungen im aktuellen Suchpfad zu speichern, und so Zyklen im Ablauf der Suche zu erkennen. Dies kann entweder über einen weiteren Flag in der Transposition Table erfolgen, oder aber man verwendet einen eigenen Hashtable zu diesem Zweck. Man kann Hashtables auch benutzen, um Ergebnisse der Evaluationsfunktion (wie das Gesamtergebnis, Informationen über die Bauernstruktur oder über die Königssicherheit) zu speichern und später wiederverwenden zu können. Diese Einsatzmöglichkeiten sind eigentlich nur durch die Kapazität des zur Verfügung stehenden Speichers begrenzt.

2.2.4. Vorsortierung der Schlagzüge (MVV/LVA)

Wie bereits dargelegt, werden die möglichen Züge vorsortiert, da die frühe Erzeugung guter Ergebnisse zu mehr Alpha/Beta-Schnitten im weiteren Verlauf der Suche führt. Es werden alle die

11 Der Fall von Inkonsistenzen durch Überinformation ist auch möglich, wird jedoch vernachlässigt

Züge als gut betrachtet, die ein möglichst hohes Ergebnis für den aktiven Spieler ermöglichen. Diese hohen Werte sind im Schach vor allem durch **Materialgewinn** zu erreichen. Wenn eine gegnerische Figur geschlagen wurde, so hat dies einen massiven positiven Einfluß auf die Bewertung der Stellung. Es ist also logisch, alle möglichen Schlagzüge als erste zu untersuchen, da die Wahrscheinlichkeit hoch ist, daß sie zu einer Veränderung der Bewertung führen; dies ist nur dann nicht der Fall, wenn der Gegenspieler das Ergebnis durch einen weiteren Schlagzug ausgleichen kann.
Die einfachste Möglichkeit der Vorsortierung von Schlagzüge ist das MVV/LVA – Schema (Most Valuable Victim/Least Valuable Agressor). Hierbei wird angenommen, daß es Priorität hat, Figuren mit möglichst hohem Wert zu schlagen, um eine möglichst große positive Veränderung des Ergebnisses zu erzielen. Die Schlagzüge werden also nach diesem Kriterium sortiert. Sollte es mehrere Schlagzüge mit gleich hohem Wert geben, so werden diese nach dem Wert der schlagenden Figur sortiert, diesmal in aufsteigender Reihenfolge. Hier liegt die Annahme zu Grunde, daß es besser ist, mit einer kleinen Figur zu schlagen als mit einer großen, da so mögliche Verschlechterungen des Ergebnisses minimiert werden können, falls der Gegner seinerseits mit einem Schlagzug kontern kann.
Die Vorsortierung nach dem MVV/LVA – Kriterium findet in jedem Schachprogramm Anwendung, da sie eine einfache Möglichkeit darstellt, einen Teil der möglichen Züge nach ihrem zu erwartenden Gewinn zu sortieren.

2.2.5. Vorsortierung der Schlagzüge (SEE)

Eine weitere Möglichkeit, Schlagzüge zu sortieren, bietet die Static Exchange Evaluation (SEE). Dieses Verfahren evaluiert den möglichen Gewinn eines Schlagzuges, indem alle möglichen folgenden **Schlagkombinationen** auf dem Zielfeld berechnet werden. Hierzu werden alle Figuren ermittelt, die einen Schlagzug auf dieses Feld ausführen können. Dann wird der sog. **Swapoff-Algorithmus** [Levy 76] ausgeführt, der angibt, welches Ergebnis bei optimaler Auswahl der schlagenden Figuren beider Seiten möglich ist. Dies bedeutet, daß angenommen wird, daß beide Spieler immer zuerst mit den niedrigwertigsten Figuren schlagen werden. Die möglichen Schlagzüge werden nach diesem Kriterium sortiert und alternierend ausgeführt. Außerdem muß berücksichtigt werden, daß jeder Spieler die Möglichkeit hat, die Schlagfolge abzubrechen, wenn er sich durch weitere Schlagzüge nur verschlechtern kann. Sind nun alle möglichen Schlagabfolgen berücksichtigt, so gibt das Ergebnis der SEE den theoretischen Gewinn (oder Verlust) für den Spieler an, der den ersten Schlagzug ausgeführt hat.
Dieses Verfahren besitzt allerdings gewisse Einschränkungen: Da die Schlagkombinationen für einzelne Felder isoliert betrachtet werden, können unrealistische Werte entstehen. So werden z.B. Fesselungen nicht berücksichtigt, also die Möglichkeit, daß eine Figur nicht ziehen darf, weil sie sonst den eigenen König ins Schach stellen oder eine andere wertvolle Figur entblößen würde. Auch ist nicht berücksichtigt, daß Figuren nicht gleichzeitig an zwei Schlagfolgen auf verschiedenen Feldern teilnehmen können, denn würde die Figur in der einen Kombination tatsächlich geschlagen, stünde sie für eine andere natürlich nicht mehr zur Verfügung.
Dieses Faktoren führen zu einer Ungenauigkeit der Ergebnisse, trotzdem kann das Ergebnis der SEE benutzt werden, um die Sortierung der Schlagzüge nach dem zu erwartenden Gewinn zu optimieren. Dieses Verfahren ist genauer als das oben gezeigte MVV/LVA – Schema (da nicht nur der mögliche Gewinn des ersten Schlagzuges berücksichtigt wird), jedoch auch mit deutlich mehr Aufwand verbunden: Eine einfache Implementation erfolgt rekursiv, was aber für eine praktische Anwendung nicht effizient genug ist. Es gibt auch die Möglichkeit, den Algorithmus mit verketteten Listen zu implementieren, dies ist jedoch nicht trivial und erfordert ein sehr präzises Verständnis des Ablaufs.

2.2.6. Vorsortierung durch Iterative Deepening

Die Reihenfolge der Knotenexpansionen ist, wie in 2.1.5. gezeigt, ein ausschlaggebender Faktor für die Effizienz aller Schnittverfahren. Zur Unterstützung der Vorsortierung von Knoten bietet sich ein in der Schachprogrammierung bekanntes Verfahren namens Iterative Deepening an. Hierbei wird nicht zu einer festen Suchtiefe gesucht, stattdessen wird die Tiefe der Suche an der Wurzel **sukzessive** erhöht, bis die gewünschte Suchtiefe erreicht ist.
Iterative Deepening ist effizienzsteigernd, da in geringen Suchtiefen mit wenig Aufwand Ergebnisse gesammelt werden, die später zur Beschleunigung der Suche verwendet werden können. So wird der Aufwand einer tieferen Suche, der exponentiell wächst, deutlich reduziert. In der Praxis wird man die bei der vorhergehenden Suche gefundene Hauptvariante (die Abfolge der bestmöglichen Züge beider Spieler) benutzen, um in der folgenden Suche diese Hauptvariante als erste zu traversieren und von ihr ausgehend neue Ergebnisse zu finden. Dies geschieht am einfachsten über die Speicherung des besten Zuges der betreffenden Knoten in der Transposition Table. Der hierdurch gewonnene Zuwachs an Effizienz des Suchverfahrens wiegt den Overhead der wiederholten Suche auf: Tatsächlich ist eine Suche mit Iterative Deepening und Transposition Table über die Suchtiefen 1, 2, 3 ... n schneller als eine einzige Suche bis zur Tiefe n [Marsland 83].
Auch die Verwendung eines Aspiration Window profitiert von Iterative Deepening, da das Ergebnis einer Suchtiefe n als realistischer Mittelwert für das Aspiration Window der Suchtiefe n+1 angenommen werden kann. So wird die Gefahr möglicher Wiederholungssuchen aufgrund eines Aspiration Window Fail minimiert. Zur Beschleunigung des MTD(f)-Algorithmus kann bei Verwendung des Iterative Deepening - Verfahrens das Ergebnis der vorherigen Suchtiefe als realistischer Wert für das Nullfenster der neuen Suchtiefe verwendet werden. In beiden Fällen liegt die Annahme zu Grunde, daß es wahrscheinlich ist, daß das Ergebnis aus einer neuen Suchtiefe in ähnlichen Bereichen liegt wie das aus der vorherigen.
Ein weiterer Vorteil des Verfahrens ist die Möglichkeit, den Zeithorizont einer Suche gut kontrollieren zu können. Man kann nach Ablauf des Zeitlimits die Suche nach Vollendung der aktuellen Suchtiefe abbrechen, da bereits ein verwertbares Ergebnis aus der vorherigen Suche vorliegt. Da der Aufwand einer Suche a-priori nicht bekannt ist, erlaubt dieser Ansatz eine höhere Granularität im Einsatz der vorhandenen zeitlichen Ressourcen. Dieser Faktor ist sehr wichtig, wenn Spiele innerhalb eines Zeitlimits ausgetragen werden.
Außerdem ist es auch möglich, Iterative Deepening nicht nur an der Wurzel zu verwenden. Das sog. Internal Iterative Deepening (IID) geht davon aus, daß es effizienter ist, innerhalb des Suchbaums mit einer flachen Suche eine besten Zug zu bestimmen und diesen dann als ersten tiefer zu untersuchen, als ohne gute Vorsortierung, eine tiefe Suche durchzuführen. Dieses Verfahren wird in der Praxis immer dann angewendet, wenn die verbleibende Suchtiefe größer zwei ist (also ein Informationsgewinn durch eine flachere Suche überhaupt möglich ist) und kein bester Zug aus der Transposition Table zur Verfügung steht. Dieser Ansatz bringt leichte zusätzliche Performanzgewinne.

2.2.7. Vorsortierung durch History/Killer Heuristic

Die History und die Killer Heuristic sind Verfahren zur Verbesserung der Vorsortierung von Knoten. Es wird davon ausgegangen, daß Züge, die sich bei vorhergehenden Suchen als gut herausgestellt haben, dies bei weiteren Suchen ebenfalls tun werden. Diese Züge werden gespeichert, und falls sie erneut erzeugt werden, werden sie als erste untersucht. Hierbei handelt es sich um eine Heuristik, da keinesfalls sichergestellt ist, daß diese Züge immer noch gut sind: Wenn bei einer folgenden Suche eine größere Suchtiefe erreicht wird, so kann es sein, daß ein Zug, der bei geringerer Suchtiefe gut war, sich nun als schlecht herausstellt. In der praktischen Anwendung

treten jedoch weit mehr Fälle auf, in denen die Effizienz des Suchvorgangs durch die Anwendung dieser Heuristiken erhöht wird. Vor allem in Verbindung mit Iterative Deepening sind diese Verfahren sehr interessant: Da hier von der Wurzel aus mehrere Suchen mit unterschiedlicher Suchtiefe betrieben werden, ist es sicher, daß viele Knoten, gerade der oberen Suchebenen, mehrmals untersucht werden. Die Suche kann hier durch Vorsortierung, basierend auf den Informationen der Heuristiken, beschleunigt werden.

Die **Killer Heuristic** geht nach einem einfachen Schema vor: Zu jeder Suchebene werden zwei (implementationsabhängig auch drei) Züge gespeichert, die zu dem gegebenen Zeitpunkt die höchsten Ergebnisse erreicht haben. Sollten diese Züge nun in weiteren Positionen dieser Suchebene erzeugt werden, so werden sie in der Zugliste nach vorne sortiert, um möglichst früh ausgeführt zu werden. Dieser Ansatz versucht aus **lokalen** Informationen (da für jede Suchtiefe andere Daten gespeichert werden) die Vorsortierung der Züge zu verbessern.

Die **History Heuristic** verwendet einen **globalen** Ansatz. Es wird eine 64*64 Feld Array benutzt, um jeden möglichen Zug anhand seines Anfangs- und Zielfelds bestimmen zu können. Nun wird für jeden Zug, der einen Beta-Schnitt verursacht oder sich als bester Zug herausstellt, der zugehörige Zähler in diesem Array inkrementiert. Diese Zähler werden benutzt, um Züge mit hohen Werten bei erneuter Erzeugung in der Zugliste nach vorne zu sortieren. Der Unterschied zur Killer Heuristic liegt darin, daß in der History Heuristic alle Züge über den gesamten Suchbaum bewertet werden, wohingegen die Killer Heuristic nur ausgewählte Züge der gleichen Suchebene berücksichtigt.

Alle Suchalgorithmen, die Alpha/Beta-Schnitte verwenden, können von diesen Heuristiken profitieren. Durch die frühe Erzeugung guter Züge wird die Menge möglicher Schnitte erhöht und dadurch die Suche beschleunigt. Außerdem kann durch ihre Anwendung die Anzahl notwendiger Wiederholungssuchen verringert werden.

2.2.8. Reduzierung des Suchraums durch Verified Nullmove Pruning

Nullmove Pruning ist ein Verfahren, das heutzutage in jedem Schachprogramm Anwendung findet. Die ersten theoretischen Schriften hierzu sind [Beal 89 / Goetsch; Campbell 90]. Das Verfahren basiert auf folgender Heuristik: Man nimmt an, daß es für den aktiven Spieler eine Zugmöglichkeit gibt, die mindestens das gleiche Ergebnis liefert wie die, den Gegner zweimal hintereinander ziehen zu lassen. Die Annahme scheint sinnvoll, da im Schach das Zugrecht ein großer Vorteil ist, da man die Möglichkeit hat, durch einen eigenen Zug die Stellung positiv zu beeinflußen. Die **Abgabe des Zugrechts** stellt also in den allermeisten Fällen einen Nachteil dar, es sollte immer möglich sein, mit einem normalen Zug das Ergebnis zu erreichen, das bei Abgabe des Zugrechts, Nullmove genannt, möglich ist. Es gibt jedoch Situationen, in denen diese Heuristik nicht zutrifft, worauf ich später noch eingehen werde.
In einer praktischen Anwendung des Verfahrens wird man vor Ausführung der eigenen Züge eine Suche starten, die dem Gegenspieler die Möglichkeit gibt, erneut zu ziehen, dies ist die **Nullmove-Suche**. Diese Suche führt man zu einer reduzierten Tiefe aus, die Tiefenreduktion beträgt dabei entweder zwei oder drei Halbzüge. Sollte diese Nullmove-Suche einen Beta-Cutoff verursachen, so kann man aufgrund der obigen Annahme an dieser Stelle die Suche abbrechen, da man davon ausgeht, daß ein mindestens gleich gutes Ergebnis auch durch eine normale Suche erreicht werden kann. Man hat also mit dem geringen Aufwand einer Suche zu reduzierter Tiefe einen Schnitt verursacht, was aufgrund des exponentiellen Verhältnisses zwischen der Erhöhung der Suchtiefe und der Steigerung des Aufwands sehr viel schneller ist, als zur vollen Tiefe zu suchen. Aus diesem Grund wird Nullmove Pruning (trotz möglicher Risiken) verwendet, da es sehr deutliche Geschwindigkeitsverbesserungen in der Suche mit sich bringt.

Es gibt aber auch Situationen, in denen die Annahme des Nullmove Pruning fehlerhafte Ergebnisse erzeugt, diese Situationen nennt man im Schach **Zugzwang**. Zugzwang bedeutet, daß man sich durch eigene Züge nicht verbessern kann, also die Möglichkeit, das Zugrecht abzugeben, einen Vorteil darstellt. Diese Situationen sind im Mittelspiel relativ selten, treten aber vermehrt in Endspielen auf. In diesen Situationen wird Nullmove Pruning die Ergebnisse deutlich verfälschen, da die Grundannahme der Heuristik nicht erfüllt ist.
Um dieser Problematik zu entgehen, wurde das sog. Verified Nullmove Pruning entwickelt [Tabibi 02]. Dieses Verfahren geht von der gleichen Annahme aus, verfährt jedoch anders mit den Ergebnissen: Sollte mit der Nullmove-Suche (Tiefenreduktion $r = 3$) ein Beta-Schnitt verursacht werden können, so wird nicht sofort abgeschnitten. Stattdessen wird für die eigentliche Suche der Kindknoten die Suchtiefe reduziert ($r = 1$ oder $r = 2$). Diese Suche führt eine Verifikation des Beta-Schnitts der Nullmove-Suche durch; sollte innerhalb dieser Suche ein Nullmove einen Beta-Schnitt verursachen, so kann abgeschnitten werden.
Ist die Suche zur reduzierten Tiefe abgeschlossen, so wird geprüft, ob mit dem Ergebnis tatsächlich ein Beta-Schnitt verursacht werden kann, das Ergebnis der Nullmove-Suche also bestätigt wurde. Ist dies der Fall, so hat man durch die Reduzierung der Suchtiefe und mögliche weitere Nullmove-Schnitte in Unterbäumen die Suche entscheidend beschleunigt. Das Ergebnis ist dabei als sicher anzunehmen, da die Tiefenreduktion in den Kindknoten kleiner ist als die der anfänglichen Nullmove-Suche, die Verifikation des Nullmove Beta-Schnitts also den Beweis liefert, daß die Annahme der Heuristik richtig war. Ist der Beta-Schnitt hingegen nicht möglich, so handelt es sich um eine Zugzwang-Situation, in diesem Fall muß die Suche zur vollständigen (nicht reduzierten) Tiefe **wiederholt** werden.
Verified Nullmove Pruning versucht auf die dargestellte Art und Weise, die Fehlermöglichkeiten des einfachen Nullmove Pruning auszuschließen. Die mögliche Wiederholung der Suche ist zwar nachteilig, die Reduzierung der Gesamttiefe sowie die Sicherheit der Ergebnisse machen es aber trotzdem zum sowohl schnelleren als auch sichereren Verfahren. Wie Verified Nullmove Pruning in Nemesis implementiert wurde und wie man das Verfahren weiter optimieren kann, ist in 4.4.3. dargestellt.

2.2.9. Effizienzsteigerung durch Futility Pruning

Futility Pruning ist ein Verfahren, das erstmals in [Schaeffer 86] erwähnt wurde. Es basiert auf der in 2.1.11. dargestellten Möglichkeit des Stand-Pat – Beta-Schnitt innerhalb der Quiescence Search. Es kann auf verschiedene Weise implementiert werden: Es gibt sowohl die Möglichkeit, Futility Pruning als rein effizienzsteigernde Maßnahme, also ohne Qualitätsverschlechterung der Ergebnisse, einzusetzen, als auch einen heuristischen Anteil zu integrieren, welcher die Ergebnisse spekulativ macht, aber noch mehr Schnitte verursachen kann.
Futility Pruning wird an Knoten einen Halbzug vor dem Suchhorizont sowie innerhalb der Quiescence Search eingesetzt, also überall da, wo auf der nächsten Suchebene ein Stand-Pat Beta-Schnitt möglich ist. Die Anzahl der Knoten wächst exponentiell mit jeder Suchtiefe, die Menge dieser Knoten, die sich alle in größtmöglicher Tiefe befinden, ist also nicht zu unterschätzen.
Da bekannt ist, daß auf der nächsten Suchebene ein Stand-Pat Beta-Schnitt durchgeführt werden kann, verfährt Futility Pruning nun nach dem folgende Ansatz: Ist der Materialwert nach Ausführung eines Zuges **plus** einem angenommenen Maximalwert der positionellen Bewertung **kleiner** gleich Alpha, so wird der entstehende Knoten in der nächsten Suchebene einen Stand-Pat Beta-Schnitt verursachen. Das angenommene Ergebnis entspricht nämlich genau dem Wert, den der andere Spieler bei schlechtestmöglicher Evaluation auf der nächsten Ebene erreichen könnte. Ist dieser Wert dort größer gleich Beta (was einem Wert kleiner gleich Alpha auf der vorherigen Ebene entspricht), so ist es sicher, einen Stand-Pat Beta-Schnitt anzunehmen, ohne den wahren Wert der Evaluation zu kennen. Man kann diesen Zug bereits eine Ebene früher abschneiden, da sein

Ergebnis außerhalb des Suchfensters liegen wird; in einer praktischen Anwendung wird man diesen Knoten von der Untersuchung der Kindknoten ausschließen.
Futility Pruning kann grundsätzlich nicht angewendet werden, wenn der aktive Spieler im Schach steht, da ansonsten irreguläre Ergebnisse erzeugt würden. Außerdem wird es erst dann verwendet, wenn schon mindestens ein legaler Zug ausgeführt wurde, ansonsten bestünde die Möglichkeit, eine Suche zu vollenden, ohne einen einzigen Kindknoten untersucht zu haben, was irreguläre Ergebnisse erzeugen würde.
Die Qualität der Ergebnisse hängt nun entscheidend von dem für die Evaluation angenommenen Wert ab. Wird hier ein Wert verwendet, der dem tatsächlichen Maximalwert der Evaluation entspricht, so ist Futility Pruning in dem Sinne sicher, daß es keinen Einfluß auf das Suchergebnis haben kann. Wenn man diesen Wert als Heuristik verwendet, so kann man ihn gegenüber dem möglichen Maximum reduzieren. Dies führt zu einem Geschwindigkeitszuwachs, da noch mehr Knoten abgeschnitten werden können. Die Ergebnisse sind aber nicht mehr sicher, da die theoretische Möglichkeit besteht, daß der wahre Wert der Evaluation größer ist als der angenommene, so daß auf der nächsten Suchebene kein Stand-Pat – Beta-Schnitt möglich wäre; auf diese Weise können fehlerhafte Ergebnisse erzeugt werden. Dieser Effekt ist schwierig zu kontrollieren, bietet aber gleichzeitig Raum zur Verbesserung der Suchgeschwindigkeit.

2.2.10. Effizienzsteigerung durch Lazy Evaluation

Lazy Evaluation ist eine Möglichkeit, den Ablauf der Quiescence Search zu beschleunigen. Wie der Name bereits andeutet, versucht man den Aufwand einer vollständigen Evaluation eines Knotens zu vermeiden. Dies geschieht auf Basis der folgenden Annahme: Da innerhalb der Quiescence Search der Stand-Pat - Beta-Schnitt durchgeführt wird, muß hierfür der Evaluationswert jedes besuchten Knotens berechnet werden. Dieser Evaluationswert besteht aus dem aktuellen Materialwert sowie einer Bewertung der positionellen Faktoren der Stellung. Ein Stand-Pat – Beta-Schnitt ist jedoch bereits dann als sicher anzunehmen, wenn bekannt ist, daß der aktuelle Materialwert **minus** dem maximalen Wert der positionellen Bewertung **größer** gleich Beta ist. In diesem Fall ist es nicht notwendig, den exakten Wert der Evaluation zu berechnen.
Da der Materialwert inkrementell gehandhabt wird (s. 4.4.1.), ist er mit minimalem Aufwand verfügbar. Sollte es möglich sein, aufgrund der dargestellten Annahme einen Beta-Schnitt zu verursachen, so hat man den sehr viel größeren Aufwand einer positionellen Evaluation gespart. Das Verfahren kann jedoch nicht angewandt werden, wenn der aktive Spieler im Schach steht, weil dann kein Stand-Pat – Beta-Schnitt möglich ist.
Lazy Evaluation hat einen ähnlichen Ansatz wie Futility Pruning, deswegen ist es auch hier möglich, durch Anpassung des für die positionelle Evaluation angenommenen Maximalwertes sowohl sichere als auch spekulative Ergebnisse zu erzeugen. Die Art der Implementation dieser beiden Verfahren kann 4.4.3. sowie 4.4.3. entnommen werden.

2.2.11. Verwendung einer Eröffnungsbibliothek

Die verschiedenen Möglichkeiten, eine Schachpartie zu eröffnen, sind in der Geschichte des Schachspiels sehr ausführlich analysiert worden. Die Eröffnung stellt hohe Anforderungen an den Spieler, da hier die Ausgangsposition für mögliche Entwicklungen im Mittelspiel bestimmt wird. Einfache Fehler in der Eröffnung können verheerende Folgen haben, da sie zu einem positionellen Nachteil führen, der im Mittelspiel immer schwieriger auszugleichen wird. Gerade ein Schachprogramm tut sich schwer, in dieser Phase des Spiels die korrekten Züge zu finden, da sie oft Teil eines größeren Plans sind, den das Programm innerhalb seines Suchhorizonts nicht überblicken kann. Deshalb ist man dazu übergegangen, Schachprogramme mit

Eröffnungsbibliotheken auszustatten, die große Mengen an verschiedenen Zugvarianten enthalten, die als optimale Eröffnungen bekannt sind.
Die Bibliothek wird vor Spielbeginn eingelesen und in einer geeigneten Datenstruktur, meist der Transposition Table, gespeichert. Sobald das Spiel beginnt, wird zuerst geprüft, ob die aktuelle Ausgangsposition in dieser Datenstruktur enthalten ist. Wenn ja, wird der hier bekannte beste Zug gespielt, ohne eine Suche durchzuführen. So ist sichergestellt, daß das Programm in dieser Phase des Spiels keine positionellen Fehler macht.
Eine Eröffnungsbibliothek ist gerade für eine Schachanwendung in ressourcenarmen Systemen wichtig, da hier davon ausgegangen werden muß, daß keine hohen Suchtiefen erreicht werden. Dies führt dazu, daß gerade strategische Entwicklungen nur schwer erkannt werden können. Die Bibliothek kann helfen, diese Schwäche in der Eröffnung zu überwinden. Die Verwendung einer Eröffnungsbibliothek hat als weiteren positiven Aspekt, daß für diese Phase des Spiels keine Suchzeit benötigt wird: Sollte im Spiel eine Standard-Eröffnung gespielt werden, so kann das Programm die Züge dieser Eröffnung aus der Bibliothek spielen, ohne Zeit für eine Suche zu verbrauchen. Je nach Größe der Bibliothek kann hier die Zeit für mehr als zehn MinMax-Suchen eingespart werden.
Die Implementation in Nemesis sowie die verwendete Eröffnungsbibliothek werde ich in 4.5.1. genauer beschreiben.

3. Zielplattformen für Schachprogramme

3.1. Geschichtlicher Rückblick

Die Schachprogrammierung hat sich seit den ersten theoretischen Schriften der 50er Jahre enorm entwickelt. Das erste Programm, das großes Aufsehen erregt hat, war Chess 4.6 von David Slate und Larry Atkin. Die Entwicklung dieses Programms geht bis auf das Jahr 1968 zurück, als eine Gruppe von jungen Wissenschaftlern an Chicagos Northwestern University begann, sich für Schachprogrammierung zu interessieren. Der Ingenieurstudent Larry Atkin und der Diplomphysiker David Slate schrieben eine Serie von Schachprogrammen, die bis 1972 eine spielstarke Version, Chess 3.6, hervorbrachte. Für die Computerschach-Weltmeisterschaften 1973 beschlossen Slate und Atkin, ihr Programm völlig umzuschreiben, vor allem um eine brute force Tiefensuche bis zu einer festen Maximaltiefe, meist von fünf Halbzügen, zu implementieren. Außerdem wurde in diesem Programm eine Quiescence Search, wie sie heute üblich ist, verwendet [Slate; Atkin 77]. Es wurden also erstmals alle die Techniken benutzt, die grundlegend für die hohe Spielstärke moderner Schachprogramme sind. So entstand ein Programm, das allen anderen Programmen überlegen war und erstmals vermochte, menschliche Schachmeister vor Probleme zu stellen.
Chess 4.5, wie die 1976 vorgestellte Version genannt wurde, gewann vier US-Computerschach-Meisterschaften hintereinander. Dies war bereits dem Vorgänger Chess 3.6 gelungen [Wall]. Erstmals wurden auch Erfolge bei klassischen Turnieren verbucht, im gleichen Jahr gewann Chess 4.5 das Paul-Masson-Turnier in Saragota, Californien, in dem es alle fünf menschlichen Gegner schlug [Ars Electronica]. Bei der zweiten Computerschach-Weltmeisterschaft 1977 in Toronto siegte dann die nochmals leicht verbesserte Version Chess 4.6; im gleichen Jahr gelang diesem Programm der erste Sieg eines Computers gegen einen menschlichen Großmeister, was zur damaligen Zeit einen Durchbruch darstellte (s.u.).

Der Erfolg von Chess 4.6 basiert nicht zuletzt auf der **Großrechenanlage** CYBER 176 von Control Data: Geräte dieser Serie waren die schnellsten verfügbaren Rechner der damaligen Zeit. Die CPU hatte eine Taktfrequenz von maximal 40 Mhz, sie war jedoch zusätzlich mit bis zu 20 Peripherie-Prozessoren ausgestattet, die für alle Ein- und Ausgabefunktionalitäten zuständig waren. Die Peripherie-Prozessoren verwendeten eine „Barrel and Slot“ genannte Technik, um Anweisungen der CPU auszuführen. Jeder Prozessor hatte hierzu eigenen Speicher und eigene Register, aber nur die CPU selbst konnte Anweisungen sukzessive weitergeben und tatsächlich zur Ausführung bringen, was eine sehr primitive Form von Hardware-Multiprogramming darstellt. Die Prozessoren hatten einen internen Datenbus von 60-Bit, der in Einheiten von sechs mal zehn Bit zerlegt wurde. Die CPU hatte einen Cache von 2 Kilobyte und verwendete jeweils acht Register für Adressen, Anweisungen und Indizes. Die Systeme der Cyber 176 Serie waren mit maximal 512 Kilobyte Ram bestückt [Wikipedia Cyber]. Maschinen dieser Serie hatten eine interne Verarbeitungsgeschwindigkeit von ca. 18 Millionen Befehlen pro Sekunde, damit konnte Chess 4.6 in den 180 Sekunden, die bei einer Turnierpartie pro Zug zur Verfügung stehen, über 700.000 Stellungen bewerten [Ars Electronica].

Die folgende Abbildung zeigt das Nachfolgemodell Cyber 180:

Abb. 3.1: Großrechenanlage Cyber 180 [Cray-Cyber Team]

Computerschach profitierte davon, daß in anderen Bereichen der Forschung immer höhere Rechenleistungen erforderlich wurden, da immer komplexere Berechnungen in möglichst kurzer Zeit durchführbar sein sollten. Die Verwendung eines CYBER 170 für Chess 4.6 ist ein gutes Beispiel für die Auswahl einer Zielplattform für eine Schachanwendung: Die CYBER 170 Serie wurde konzipiert, um in Bereichen der wissenschaftlichen Analyse sowie für militärische Zwecke verwendet zu werden, stellte aber gleichzeitig die optimale Plattform für ein Schachprogramm dar, da sie die für damalige Zeit größtmögliche Performanz erzielte.

Chess 4.6 hat vor allem deshalb soviel Aufmerksamkeit erregt, weil es lange als ausgeschlossen galt, daß ein Schachprogramm gegen einen menschlichen Großmeister gewinnen kann. Dies wurde selbst in Fachkreisen immer wieder bezweifelt, da man es für ausgeschlossen hielt, daß irgendwann die hierfür notwendige Rechenleistung tatsächlich zu erbringen sei. Noch im August 1968 hatte der Internationale Meister David Levy um 1250 Pfund Sterling gewettet: "Mich wird 10 Jahre lang kein Computerprogramm bezwingen" [Computerwoche]. Doch 1977 war es dann soweit, daß Chess 4.6 gegen den englischen Großmeister M. Stean gewann und 1978 dann auch ebenfalls gegen David Levy. Dies wurde als außerordentlicher Durchbruch angesehen, da es zeigte, daß Computerschach auch auf höchstem menschlichem Niveau bestehen kann. Hierzu waren damals teure Großrechenanlagen, wie die Cyber 176, notwendig. Warum sich diese Rahmenbedingungen deutlich geändert haben, wird im nächsten Abschnitt ersichtlich.

3.2. Aktuelle Systeme

Seit den 70er Jahren hat die Mikroprozessortechnologie erstaunliche Fortschritte gemacht. Die große Verbreitung des Personal Computers hat dazu beigetragen, technische Entwicklungen zu beschleunigen und einer breiten Öffentlichkeit zugänglich zu machen. Moderne PCs haben ein Vielfaches der Taktfrequenz und Speicherausstattung damaliger Großrechenanlagen und können sehr viel höhere Rechenleistungen erbringen. Gleichzeitig sind sie für jedermann verfügbar, da es sich nicht mehr um hochspezialisierte Geräte für wissenschaftliche Anwendungen handelt. Dies gilt auch für den Bereich mobiler Geräte, die im Vergleich zu stationären Systemen über deutlich weniger Ressourcen verfügen. Tatsächlich aber sind diese Geräte schon soweit fortgeschritten, daß sie die Leistungsfähigkeit der im letzten Kapitel vorgestellten Systeme deutlich übertreffen.
So hat sich in den letzten Jahren die technische Entwicklung vor allem im Bereich der **PDAs** stark beschleunigt. Es sind heutzutage Geräte auf dem Markt, deren Prozessoren mit bis zu 624 Mhz getaktet sind und die mit 128 Megabyte Ram ausgestattet werden [Chip.de]. Die hiermit erreichbare Rechenleistung entspricht in etwa dem Zehnfachen von dem, was mit früheren Großrechnern, wie dem Cyber 176, möglich war. Diese PDAs sind aber nun keinesfalls spezialisierte Rechenanlagen, im Gegenteil. Sie sind vor allem für Zwecke der Datenverwaltung, Terminplanung und als Unterhaltungsmedien gedacht. Da hierzu teilweise auch eine hohe Rechenleistung erforderlich ist, ist diese eigentlich nur ein Nebeneffekt, nicht das tatsächlich Aufgabengebiet dieser Geräte. Sie sind für den Konsumentenmarkt bestimmt und befinden sich deshalb in Preisregionen, die auch für Einzelpersonen zugänglich sind.
Noch deutlicher wird diese Entwicklung, wenn man Geräte der **Smartphone** - Klasse betrachtet. Hiermit bezeichnet man mobile Telefone, die zur Ausführung von Programmen mit einer programmierbaren CPU sowie einer eigenen Speicherumgebung ausgestattet werden. Die Entwicklung dieser Geräte geht den gleichen Weg wie die Entwicklung von PDAs, aktuelle Smartphones haben bereits eine CPU mit 100 bis 150 Mhz Systemtakt. Sie werden mit 4 bis 8 Megabyte Hauptspeicher und bis zu 64 Megabyte Festspeicher ausgestattet [Futuremark]. Die Firma HTC vertreibt Smartphones [HTC], die Microsoft Windows Mobile Edition unterstützen, dieses Betriebssystem wird aktuell vor allem von PDAs genutzt und ist im Rahmen dieser Arbeit zur Ausführung von Nemesis zum Einsatz gekommen (s. 4.2.2.1.). Es gibt also auf dem Markt bereits Geräte der Smartphone-Klasse, auf denen man die von mir entwickelte Schachanwendung ebenfalls ausführen könnte.

3.3. Zukünftige Entwicklung

Wie im vorherigen Kapitel dargestellt, sind Systeme, die man im heutigen Vergleich als ressourcenarm bezeichnet, bereits in der Lage Großrechner der 70er Jahre in den Schatten zu stellen. Doch wie sieht die zukünftige Entwicklung aus? Diese Frage ist vor allem für den Bereich der Smartphones interessant: Mobile Telefone haben weltweit eine sehr hohe Verbreitung erreicht, die Größe dieses Marktes wirkt wie ein Katalysator auf die technische Entwicklung. Im Fortschritt dieser Entwicklung wird irgendwann der Punkt erreicht, wo Smartphones ähnliche Ressourcen besitzen wie heutige PDAs. Da sie jedoch bei höherer Rechenleistung die gleichen Aufgaben erfüllen könnten wie diese, so wäre es durchaus denkbar, daß PDAs irgendwann überflüssig werden. Diese Annahme scheint nicht übertrieben, da bereits Smartphones mit bis zu 200 Mhz Systemtakt entwickelt werden. Diese Geräte sind mit hochintegrierten ARM Prozessoren ausgestattet, die heutzutage bereits in vielen PDAs verwendet werden (s. 4.2.1.). Sie zeichnen sich durch niedrigen Stromverbrauch aus und sind mit allen modernen Schnittstellenkonzepten wie WLAN, Bluetooth oder USB-OTG (USB On-The-Go) ausgestattet [Freescale]. So haben sie bereits genügend Rechenleistung, um eine ganze Reihe von Anforderungen wie Internet-Zugang, Audio-/ Videowiedergabe, Terminverwaltung, Spiele etc... erfüllen zu können.
Die folgende Abbildung zeigt das Smartphone Modell Nokia 9300 [Nokia], das über 80 Megabyte internen Speicher verfügt, und Microsoft Office sowie Adobe Acrobat Formate unterstützt:

Abb. 3.2: Smartphone Nokia 9300 [Nokia]

Viele Geräte der Smartphone-Klasse werden außerdem mit eigenen Systemen zur 3D-Beschleunigung ausgestattet, was deutlich zeigt, daß sie den Anspruch haben, sich auch als Plattform für rechenintensive Anwendungen durchzusetzen [Futuremark]. Es scheint also nur eine Frage der Zeit, bis diese mobilen Telefone mit dem PDA-Segment verschmelzen und eine neue Klasse von mobilen **Allzweckgeräten** bilden. Diese Geräte werden hohe Rechenleistung mit Portabilität und vielfältigen Vernetzungsmöglichkeiten verbinden und so auf lange Sicht eine Alternative zum klassischen Personal Computer darstellen. Daß sie dabei in der Lage sein werden, hochkomplexe Aufgaben wie ein Schachprogramm zu bewältigen, scheint durch den Vergleich aktueller ressourcenarmer Systeme mit den Plattformen der Vergangenheit sehr wahrscheinlich.

4. Aufbau und Implementation der Anwendung

4.1. Anforderungskatalog

Die Anforderungen, die ich im Rahmen dieser Arbeit zu berücksichtigen habe, unterteilen sich in zwei verschiedene Kategorien: Einerseits gilt es zu beachten, welche Anforderungen die Zielumgebung erfüllen muß, um die in der Einleitung dargestellten Ergebnisse erreichen zu können. Wichtig ist hierfür, die Schachanwendung komfortabel entwickeln zu können, dies bezieht sich vor allem auf die Software-Umgebung. Die Ausführung der Schachanwendung soll möglichst hohe Suchgeschwindigkeiten erreichen, was Anforderungen an die Hardware-Umgebung stellt. Andererseits müssen auch Anforderungen an die Implementation von Nemesis 1.0 gestellt werden, um später nachvollziehen zu können, welche Ziele erreicht werden konnten. Diese Anforderungen ergeben sich aus der gewünschten hohen Qualität der zu entwickelnden Anwendung. Da in der Schachprogrammierung kein absolutes Optimum erreicht werden kann (s. 1.1.), sind diese Anforderungen teilweise nur als relative Ziele zu formulieren.

1. Anforderungen an die Zielumgebung:

1.1. Die Zielumgebung soll eine benutzerfreundliche Entwicklung des Schachprogramms ermöglichen.

1.2. Die Zielumgebung soll gute Debugging-Möglichkeiten besitzen.

1.3. Die Zielumgebung soll ein Schachspiel benutzerfreundlich darstellen können.

1.4. Die Zielplattform soll eine möglichst große Speicherumgebung zur Verfügung stellen.

1.5. Die Zielplattform soll einen möglichst hohen Systemtakt besitzen, d.h. hohe Rechenleistungen erbringen können.

2. Anforderungen an die Schachanwendung:

2.1. Die Schachanwendung soll in der Lage sein, eine Schachpartie regelkonform zu bestreiten, d.h. sie darf keine illegalen Züge ausspielen.

2.2. Die Schachanwendung soll ein stabiles Suchverhalten zeigen, das in allen (vergleichbaren) Tests die gleiche Performanz erreicht.

2.3. Die Schachanwendung soll ein korrektes Suchverhalten zeigen, also keine falschen Suchergebnisse erzeugen.

2.4. Die Schachanwendung soll in der MinMax-Suche möglichst hohe Suchtiefen erreichen.

2.5. Die Schachanwendung soll ein gutes positionelles Spiel zeigen.

2.6. Die Schachanwendung soll eine möglichst hohe Spielstärke besitzen.

Ich werde im weiteren Verlauf der Arbeit Bezug auf diese Anforderungen nehmen, um zu zeigen, welche Konsequenzen sich daraus ergeben und welchen Einfluß diese darauf haben, ob und wie die Anforderungen erfüllt werden konnten. Zusätzlich werde ich dieses in Abschnitt 6.1 nochmals zusammenfassend darstellen.

4.2. Rahmenbedingungen

4.2.1. Hardware-Umgebung

Für die praktische Durchführung meiner Arbeit hat die HAW-Hamburg mir freundlicherweise einen PDA zur Verfügung gestellt, hierbei handelt es sich um einen Hewlett Packard iPAQ Pocket PC H5500 [Hewlett Packard]. Der iPAQ H5500 ist mit einem Intel XScale 400 Mhz Prozessor ausgestattet: XScale ist eine auf Intel StrongARM basierende Chip-Mikroarchitektur, die versucht, geringen Stromverbrauch und hohe Performance miteinander zu verbinden. Die XScale-CPUs mit ARM-kompatiblem Kern werden in PDAs und Smartphones verwendet und haben alle dafür notwendigen Controller und Interfaces bereits integriert [Intel XScale]. Da ein Prozessor mit einem für mobile Geräte hohen Systemtakt zur Verfügung steht, ist die im letzen Kapitel formulierte Anforderung 1.5 durchaus erfüllt: Das Gerät ist von der Rechenleistung her gut entwickelt, seine Leistungsfähigkeit entspricht in etwa einem Personal Computer der Klasse Pentium 2 bzw. äquivalenter Prozessoren anderer Hersteller [Intel Techdocs].
Der iPAQ H5500 verfügt über 128 MB SDRAM, dieser Speicher wird vom Betriebssystem (s. 4.2.2.1.) zu gleichen Teilen in Anwendungs- und Datenspeicher unterteilt, so daß effektiv 64 MB Speicher zur Programmausführung zur Verfügung stehen. Die Größe des verfügbaren Speichers ist nach heutigen Maßstäben doch deutlich begrenzt, wodurch die Anforderung 1.4 nicht erfüllt werden kann.
Der iPAQ besitzt einen Steckplatz mit Unterstützung für SD-Speicherkarten und SDIO-Karte, die jedoch zu langsam sind, um sie bei laufender Anwendung effizient nutzen zu können. Es gäbe jedoch die Möglichkeit, zusätzliche Spielbibliotheken (s. 2.2.11.) dort zu speichern.
Das Display des iPAQ unterstützt eine Auflösung von 240 x 320 Pixel bei 64-K-Farbtiefe. Es ist also im Vergleich zu einem PC deutlich begrenzt, die Menge der darstellbaren Informationen ist gering. Bei Verwendung einer geeigneten Oberfläche (s. 4.2.2.4.) ist es aber dennoch ausreichend, um eine Schachanwendung angemessen darstellen zu können, die Anforderung 1.3 kann somit erfüllt werden. Die geringe Größe des Displays ist aber dann äußerst hinderlich, wenn größere Mengen an Debug-Informationen ausgegeben werden sollen, was eine Einschränkung der Anforderung 1.2 darstellt.
Das Gerät verfügt über eine Infrarot-Schnittstelle mit einem Datentransfer von bis zu 115,2 Kbit pro Sekunde. Desweiteren unterstützt es Bluetooth der Klasse II mit bis zu 4 dB/m Sendeleistung und einer Reichweite von etwa 10 m. Es ist also ideal geeignet für vernetzte Anwendungen, was im Rahmen eines Schachprogramms eine Möglichkeit wäre, um z.B. Menschen gegeneinander antreten zu lassen oder durch Vernetzung verschiedener Geräte eine höhere Leistungsfähigkeit zu erreichen.

Die folgende Abbildung zeigt den Hewlett Packard iPAQ Pocket PC H5500 in Originalgröße:

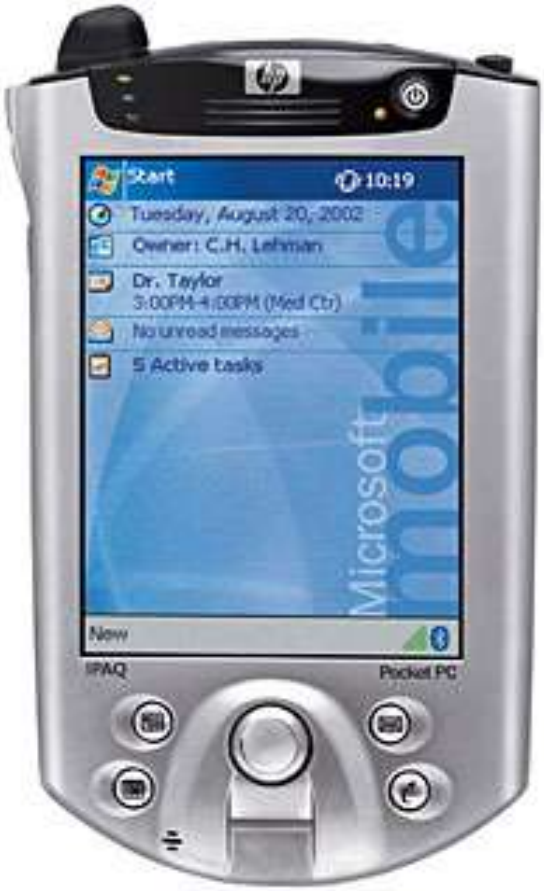


Abb. 4.1: Hewlett Packard iPAQ H5500 [Hewlett Packard]

4.2.2. Software-Umgebung

4.2.2.1. Microsoft Pocket PC 4.2

Der Hewlett Packard iPAQ H5500 wird standardmäßig mit Microsoft Pocket PC 4.2 als Betriebssystem verwendet. Pocket PC bezeichnet sowohl dieses Betriebssystem als auch eine Reihe von Geräten, die damit betrieben betrieben werden können. Pocket PC hat einen anderer Kernel als Microsoft Windows, ist also keine verkleinerte Version des Microsoft Betriebssystems für Desktop-PCs. Es läuft auf Intel x86, MIPS, ARM, und Hitachi SH Prozessoren. Pocket PC entspricht der Definition eines Echtzeitbetriebssystems mit einer festen Interrupt-Latenzzeit, es unterstützt 256 Prioritäts-Level und bietet Prioritäten-Umkehrung. Es ist außerdem Echtzeit-Multitasking-fähig, was zeigt, daß Geräte, die dieses Betriebssystem nutzen, von der Bedienbarkeit her mit Personal Computern vergleichbar sind; dies trägt zur Erfüllung der Anforderung 1.1 aus Kapitel 4.1 bei. Der Name Pocket PC wurde aus marketingtechnischen Gründen gewählt und dient dazu, sich von der Konkurrenz und dem Marktführer Palm mit dem Betriebssystem PalmOS abzugrenzen. Microsoft Pocket PC basiert auf dem Betriebssystemkern Windows CE, der für die Verwendung in Klein- und Kleinstcomputern entwickelt wurde. Pocket PC erweitert die Funktionalität von Windows CE um typische Anwendungen wie Terminkalender oder Adressverwaltung. Die Benutzeroberfläche orientiert sich dabei an derjenigen von Microsoft Windows, ist allerdings speziell für die Verwendung auf Taschencomputern angepasst worden (s. Abb. 4.1). Die aktuelle Pocket PC-

Version, die von Microsoft auch als Windows Mobile 2003 Second Edition bezeichnet wird, nutzt als Kern Windows CE .NET 4.2 [Microsoft Mobile].
Seit der Windows Mobile Version 2002 wird die Pocket PC-Plattform massiv für den Massenmarkt optimiert, was dazu führt, daß viele für den professionellen Anwender sinnvolle Funktionen, wie das Beenden von Anwendungen, Kontrolle über Netzwerkverbindungen etc... entweder unterbunden oder hinter vereinfachenden Schichten versteckt werden. Mit der Version 2002 wurde außerdem ein Connection Manager eingeführt, der die vollständige Kontrolle über jedweden Netzwerkverkehr (LAN, DFÜ, Bluetooth) übernimmt und diese Funktionalitäten automatisieren soll. Einige dieser Automatisierungen arbeiten jedoch nicht im Sinne professioneller Anwender, können aber nicht umgangen werden. Dies bedeutet eine gewisse Einschränkung der Anforderung 1.1, die auf eine benutzerfreundliche Entwicklung der Anwendung ausgerichtet ist.
Ein weiteres Problem der gesamten Pocket PC - und Windows CE - Familie sind die unterschiedlichen Prozessorarchitekturen: So ist es nicht möglich, ein Programm, das für einen CPU-Typ geschrieben wurde, auf einem anderen ausführen zu können. Zwar ist seit der Version 2002 die Pocket PC - Plattform nur noch als ARM-Variante verfügbar, aber viele alte Programme werden nicht mehr aktualisiert und stehen deshalb trotzdem nicht zur Verfügung. Die Ausführung von Anwendungen anderer Windows CE-Plattformen auf Pocket PCs ist ebenfalls aufgrund spezifischer Erweiterungen der jeweiligen Plattform kaum möglich.

4.2.2.2. Microsoft eMbedded Visual Tools 3.0

Die für die Entwicklung von Anwendungen für die verschiedenen Windows CE-Plattformen benötigten Werkzeuge und Entwicklungsumgebungen stellt Microsoft kostenlos zur Verfügung. Ich habe für meine Arbeit die Microsoft eMbedded Visual Tools 3.0 benutzt und als Entwicklungsumgebung eMbedded Visual C++ 3.0 verwendet:

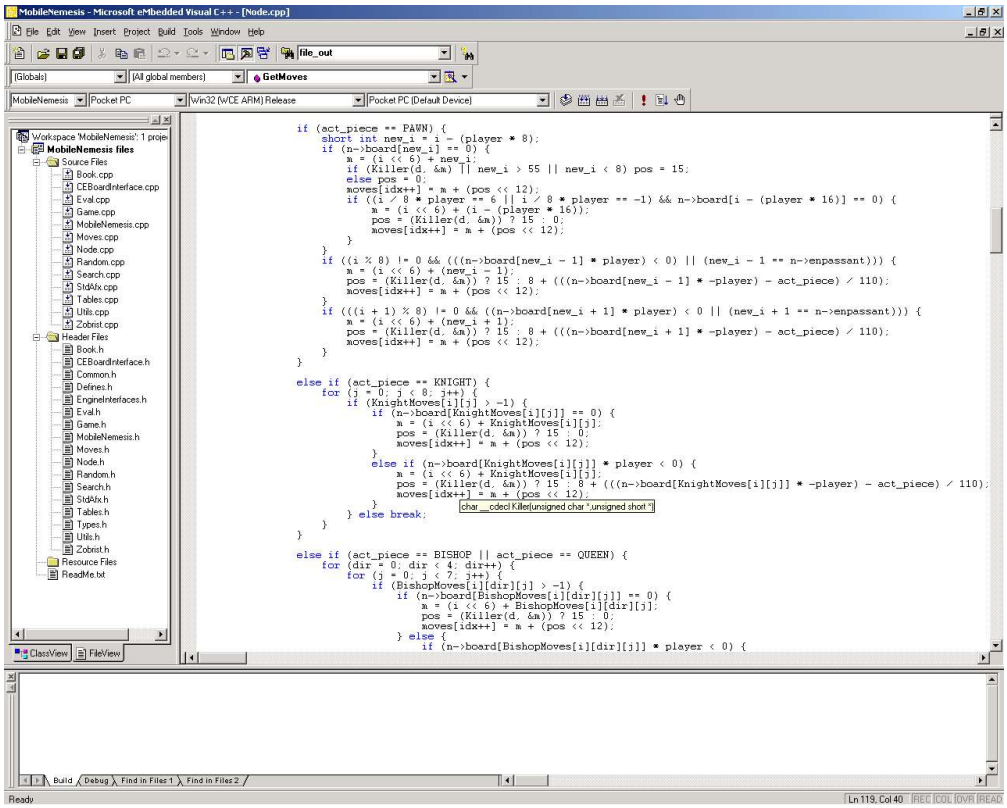


Abb. 4.2: Microsoft eMbedded Visual Tools 3.0

eMbedded Visual C++ 3.0 entspricht vom Sprachumfang her dem Microsoft Visual C++ .NET 2003 Standard, unterstützt also alle gängigen C/C++ Sprachelemente. Außerdem können MFC (Microsoft Foundation Class), ATL (Active Template Library) und das bekannte Win32 API verwendet werden. Das Win32 API inklusive MFC ist vom Funktionsumfang her beschränkter als das für Desktop-PCs, die Funktionalitäten zur Benutzerinteraktion sind jedoch fast im gleichen Umfang vorhanden wie bei der Standard Win32 API.
Ein wichtiger Unterschied von eMbedded Visual C++ 3.0 zu anderen C++ Entwicklungsumgebungen ist die Notwendigkeit mit Unicode-Strings zu arbeiten. Windows CE verwendet ausschließlich Unicode-Strings, um eine möglichst hohe Portabilität zu ermöglichen, dies führt jedoch dazu, daß, bei Benutzung der entsprechenden API, viele Typ-Konversionen erforderlich sind. Dies ist etwas unpraktisch, da der Entwickler gezwungen ist, sich mit einer Menge von über 50 Hilfsfunktionen auseinanderzusetzen, die String-Konversionen in alle verschiedenen Typen anbieten.
Eine weiteres Problem der Entwicklung mit eMbedded Visual C++ 3.0 ist, daß es kein C++ Exception-Handling unterstützt. Dementsprechend kann man mit eMbedded Visual C++ 3.0 keine Standardvorlagenbibliotheken benutzen. Auch kann man Exception Handling nicht verwenden, um Debug-Informationen zum Programmablauf auszugeben, was eine deutliche Einschränkung der Anforderung 1.2 bedeutet und in der Entwicklung extrem hinderlich ist. Diese Probleme wurden in eMbedded Visual C++ 4.0 reduziert, da Exception Handling dort implementiert wurde, es gibt jedoch noch keine Pocket PC Entwicklungsumgebung für diese Version. Außerdem benötigen, im Gegensatz zu Version 4.0, Anwendungen, die mit eMbedded Visual C++ 3.0 entwickelt wurden, keine spezielle Laufzeitumgebung, sondern werden vollständig in Maschinensprache kompiliert. Dies war für mich ein weiterer Grund, Version 3.0 zu wählen, da ein kompiliertes Programm, das keine dedizierte Laufzeitumgebung benötigt, deutlich schneller ist; dieser Faktor hat im Rahmen einer Performanzanalyse oberste Priorität.
Trotz der Verwendung von Visual C++ habe ich mich dazu entschieden, den Kern der Schachanwendung in reinem ANSI-C zu entwickeln. ANSI-C erlaubt die größte Menge an Compiler-Optimierungen und erzeugt schnelleren Code, als unter der Verwendung von C++ Sprachelementen möglich wäre. C++ wurde ausschließlich verwendet, um Ein- und Ausgabe-Funktionalitäten sowie die Kommunikation mit CEBoard (4.2.2.4.) zu implementieren.
Insgesamt erlaubt die Entwicklungsumgebung Microsoft eMbedded Visual Tools 3.0 einen komfortablen Weg, Anwendungen für einen Pocket PC oder ein vergleichbares Gerät zu entwickeln. Die oben genannten Einschränkungen sind zwar hinderlich, werden aber durch die Möglichkeit einfacher und schneller Entwicklungs- und Testzyklen ausgeglichen. Die im Anforderungskatalog formuliert Forderung nach einer benutzerfreundlichen Umgebung zur Entwicklung der Schachanwendung kann von eMbedded Visual C++ 3.0 insgesamt erfüllt werden.

4.2.2.3. Der Pocket PC Emulator

Bei allen Microsoft Pocket PC Entwicklungsumgebungen ist ein Emulator integriert, mit dem die gewünschte Zielplattform auf dem PC simuliert werden kann. Dies ermöglicht eine einfache und schnelle Entwicklung, da man den Code erst auf dem PC testen kann, bevor man ihn auf das Gerät überträgt und dort testet.
Die Entwicklung auf dem Emulator besitzt allerdings auch gewisse Einschränkungen: So kann man von der Ausführungszeit auf dem Emulator in keiner Weise auf die Performanz des Programms auf einem Pocket PC schließen. Die Pocket PC CPU und Speicherumgebung werden zwar simuliert (im Sinne ihrer Beschränkungen und der verwendeten Befehlssätze), die Ausführungszeit richtet sich jedoch maßgeblich nach dem Systemtakt, und der ist auf dem PC natürlich um einiges schneller. Außerdem unterscheiden sich Emulator und Pocket PC deutlich in ihrer Speicherverwaltung und damit in ihrer Fehlertoleranz: In der simulierten Umgebung auf dem PC werden Speicherfehler (wie

z.B. Adressierung eines Arrays außerhalb seiner Grenzen) toleriert und führen zu unbestimmbaren Resultaten im Programmablauf; auf dem Pocket PC hingegen führen sie zu einem sofortigen Programmfehler und der Notwendigkeit, daß Gerät neu zu starten, was eine Einschränkung der Anforderung nach guten Debugging-Möglichkeiten darstellt. Wenn man sich dieser Einschränkungen bewußt ist, ist der Emulator dennoch ein nützliches Werkzeug, um Anwendungen für Pocket PCs entwickeln und testen zu können, was hilft, die Anforderung 1.1 zu erfüllen.

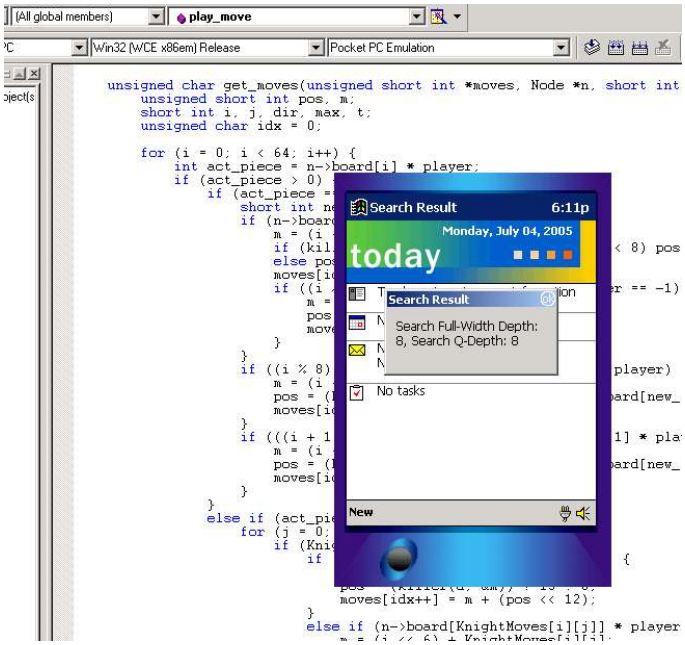


Abb. 4.3: Der Pocket PC Emulator

Wenn man direkt auf dem Gerät testen will, so kann man den iPAQ H5500 mit Microsoft ActiveSync via USB mit dem PC verbinden. Dies bietet die Möglichkeit, die Verzeichnisstruktur des angeschlossenen Gerätes direkt vom PC aus zu manipulieren, wodurch man das Programm sehr einfach auf das Gerät übertragen kann. Der einzige Unterschied zum Testen mit dem Emulator besteht darin, daß das Projekt neu kompiliert werden muß, da sich die Versionen für den Emulator und für die Zielplattform unterscheiden.

4.2.2.4. CEBoard 2.1

CEBoard ist ein Programm speziell für Pocket PCs, das entwickelt wurde, um Schachspiele auf diesen Geräten darstellen und analysieren zu können [Zanchetta]. So kann man mit CEBoard verschiedene Stellungen aufbauen und mögliche Zugkombinationen ausspielen und bewerten. Seit der Version 2.1 gibt es auch die Möglichkeit, über ein Interface mit diesem Programm zu kommunizieren, es also als **grafische Oberfläche** für ein Schachprogramm zu nutzen. Diese Möglichkeit habe ich in Anspruch genommen, da CEBoard eine sehr ausgereifte Oberfläche für eine Schachanwendung darstellt:

Abb. 4.4: CEBoard 2.1 [Zanchetta]

Die grafische Darstellung des Spielfelds, der Figuren sowie des letzten gespielten Zuges erleichtert das Verständnis der Stellung deutlich. Die Liste rechts neben dem Spielfeld enthält die im gesamten Spiel ausgeführten Züge und gibt dem Benutzer die Möglichkeit, jederzeit zu einem früheren Zustand des Spiels zurückzukehren. Unter dem Spielfeld sieht man eine Angabe über den letzten Zug sowie die Zeitkonten beider Spieler. CEBoard gibt außerdem die Möglichkeit, Informationen der Suchfunktion zum aktuellen Zustand der MinMax-Suche auszugeben: So findet man im unteren Teil des Displays das aktuelle Suchergebnis, die Suchtiefe, die für diese Suche bisher benötigte Zeit, die Anzahl der Knoten pro Sekunde sowie die Gesamtzahl der benötigten Knoten. Darunter findet man die Anzeige der Hauptvariante.
CEBoard ist durch die gute grafische Darstellung sowie die ausgereifte Funktionalität sehr gut geeignet, um als GUI für eine Schachanwendung zu fungieren, und erleichtert dem Entwickler die Aufgabe, seine Anwendung im Spielbetrieb zu testen. Die Anforderung einer benutzerfreundlichen Darstellung der Schachanwendung in der Zielumgebung kann mit Hilfe von CEBoard 2.1 voll erfüllt werden.

4.2.2.5. Chess Captor 2.21

Die Software Chess Captor 2.21 ist als Shareware im Internet verfügbar [Chess Captor]. Sie dient dazu, Mengen von Schachstellungen zu verwalten und grafisch darzustellen. Ich habe dieses Programm verwendet, um Teststellungen auf dem PC zu analysieren und Grafiken zu generieren, die ich in Kapitel 5 verwendet habe. Chess Captor bietet eine einfache Bedienung sowie einen großen Funktionsumfang.

4.3. Auswahl der Verfahren

Auf Basis der in Kapitel 4.1 dargestellten Anforderungen ist es nun wichtig zu entscheiden, wie ein Schachprogramm auf dem HP iPAQ H5500 bestmöglich zu implementieren ist. Im folgenden Kapitel werde ich erläutern, welche Verfahren ich dazu ausgewählt habe und warum ich diese für besonders geeignet halte.

4.3.1. Suchverfahren

Das Kernstück jedes modernen Schachprogramms ist die MinMax-Suche. Da es sich hier um eine Tiefensuche in einem Baum mit durchschnittlich etwa 30 Zugmöglichkeiten handelt, ist diesem Element die größte Aufmerksamkeit zu widmen. Mit Hinblick auf den exponentiellen Aufwand, der hier vom Suchverfahren geleistet werden muß, ist es am einfachsten, Ressourcen zu verschwenden oder aber Einsparungen zu erzielen.
Wie im theoretischen Teil dargestellt, ist die größte Weiterentwicklung der modernen Suchverfahren die **Nullfenstersuche**. Algorithmen, die Nullfenster verwenden, sind allgemein als die schnellsten anerkannt, wie [Marsland 91] analytisch und empirisch bewiesen hat. Es gibt zwei Algorithmen, die diesen Ansatz verwenden, sich jedoch in ihren Rahmenbedingungen deutlich unterscheiden:

Der **PVS/NegaScout-Algorithmus** hat ein ähnliches Vorgehen wie NegaMaxAlphaBeta, der einzige Unterschied ist der Versuch, durch Nullfenstersuchen die Inferiorität von Unterbäumen zu beweisen, und bei deren möglichem Fehlschlagen die notwendige Wiederholungssuche. Performanzgewinn entsteht durch jede erfolgreiche Nullfenstersuche, da sie mit einer maximalen Menge an Alpha/Beta-Schnitten verbunden ist, jede Wiederholungssuche stellt jedoch einen ernsthaften Nachteil dar, da hierfür Aufwand geleistet werden muß, der für NegaMaxAlphaBeta nicht notwendig ist. Es gibt nun zwei mögliche Ansätze, um diesen Nachteil zu minimieren: Zum einen ist die Menge der notwendigen Wiederholungssuchen zu verkleinern und zum anderen ist zu gewährleisten, daß eine Wiederholungssuche so schnell wie möglich zu einem Ergebnis führt. Für diese beiden Probleme gibt es unterschiedliche Lösungen:
Um die Menge der notwendigen Wiederholungssuchen zu verkleinern, ist es notwendig, eine gute Vorsortierung zu erreichen. Dazu gibt es verschiedene Verfahren, auf deren Auswahl ich noch eingehen werde.
Zum anderen muß der Ablauf möglicher Wiederholungssuchen betrachtet werden. Wie der Name bereits sagt, wird die Suche eines Kindknotens wiederholt, im Falle des PVS/NegaScout-Algorithmus wird das Suchfenster gegenüber der vorherigen Suche geöffnet. Dies ändert aber nichts daran, daß exakt die gleiche Menge an Knoten durchsucht werden soll wie in der Suche zuvor. Wenn also in dieser Suche eine möglichst große Menge an Informationen gespeichert wurde, so können diese Informationen benutzt werden, um die Wiederholung der Suche zu optimieren. Sowohl die Zwischenspeicherung von Ergebnissen als auch die Anwendung verschiedener Vorsortierungs-Heuristiken sind hier äußerst effizient. Für die Minimierung des zusätzlichen Aufwands sind vor allem die Verfahren zur Speicherung von Suchergebnissen unerläßlich, dies geschieht mit Hilfe von Hashtables (2.2.3.). Man kann also schließen, daß der Effizienzvorteil von PVS/NegaScout gegenüber NegaMaxAlphaBeta zu gewissen Teilen vom Vorhandensein, der Größe und der Implementation von Hashtables abhängt.

Für den **MTD(f)-Algorithmus** gelten ganz ähnliche Voraussetzungen, die Abhängigkeiten sind hier jedoch noch weitaus größer: Wie im theoretischen Teil dargestellt, basiert die Effizienz des MTD(f) nämlich ausschließlich auf Wiederholungssuchen. Einerseits ist der MTD(f)-Algorithmus in der Menge der möglichen Alpha/Beta-Schnitte nicht zu überbieten, da jede Suche eine Nullfenstersuche ist und folglich die maximale Menge an Schnitten erzeugt. Dies ist jedoch auch

der größte Nachteil dieses Verfahrens: Da die Menge der notwendigen Wiederholungssuchen nicht determiniert werden kann, hängt die Effizienz des Verfahrens völlig von der Minimierung des Aufwands dieser Wiederholungssuchen ab. Wenn die Wiederholungssuchen mit genügender Geschwindigkeit durchgeführt werden können, ist MTD(f) tatsächlich das schnellste Verfahren zur MinMax-Suche. Wie bereits für den PVS/NegaScout gezeigt, ist die Effizienz von Wiederholungssuchen abhängig von der Vorsortierung und der Geschwindigkeit der Ausführung. Die Maßnahmen zur Verbesserung der Vorsortierung sind für beide Verfahren identisch, die Ausführungsgeschwindigkeit hängt beim MTD(f) jedoch noch in sehr viel größerem Maße von den verwendeten Hashtables ab. Dies liegt vor allem daran, daß Wiederholungssuchen von der Wurzel aus durchgeführt werden, es werden also nicht, wie bei PVS/NegaScout, Unterbäume zweimal durchsucht, sondern der **gesamte** Spielbaum. Und dies nicht nur zweimal[12], sondern in der Regel etwa 5 bis 15 mal. MTD(f) ist also noch stärker auf die Möglichkeiten zur Zwischenspeicherung bereits gefundener Resultate angewiesen. Diese große Abhängigkeit von Speichermöglichkeiten, also von Hashtables, macht MTD(f) für eine Anwendung in ressourcenarmen Systemen nicht geeignet, da sie diese Anforderung nicht genügend erfüllen können.

Ich habe mich also für die Verwendung des PVS/NegaScout-Algorithmus entschieden, da er den Möglichkeiten des verwendeten Geräts am ehesten gerecht wird: Der HP iPAQ H 5500 PDA stellt zwar 64 MB RAM zur Programmausführung bereit, dies ist aber nicht ausreichend, um einen effizienten Ablauf des MTD(f)-Algorithmus garantieren zu können.

Um den verwendeten PVS/NegaScout-Algorithmus weiter zu beschleunigen, werde ich außerdem ein **Aspiration Window** verwenden, das für alle Algorithmen, die mit offenem Suchfenster starten, eine Verbesserung der Geschwindigkeit ermöglicht. Dies geschieht um der Anforderung 2.4 zu entsprechen, möglichst hohe Suchtiefen zu erreichen, was außerdem hilft, die Anforderung nach einer hohen Spielstärke zu erfüllen. Die Größe des Aspiration-Window ist implementationsabhängig, ich verwende ein Fenster der Größe +/- ein Drittel des Wertes eines Bauern, dieser Wert hat sich als bester Mittelweg zwischen der Erhöhung der Anzahl möglicher Schnitte im Suchbaum und der Minimierung der Menge an der Wurzel notwendiger Wiederholungssuchen herausgestellt.

4.3.2. Verfahren zur Vorsortierung

Wie bereits bei der Auswahl des Suchverfahrens dargestellt, sind **Hashtables** für die Geschwindigkeit außerordentlich wichtig. Sie werden aber nicht nur benutzt, um Suchergebnisse zwischenzuspeichern, sondern auch, um die Vorsortierung zu verbessern. Dies geschieht dadurch, daß zu jeder gespeicherten Position auch der bestmögliche Folgezug mitgespeichert wird. Sollte diese Stellung in einer Suche erneut erreicht werden, so kann man davon ausgehen, daß dieser Zug ein aussichtsreicher Kandidat ist, um erneut das beste Ergebnis zu liefern. Dadurch kann die Effizienz der Suche erhöht werden, was wiederum der Erfüllung der Anforderung 2.4 dient. Wenn ein bester Folgezug aus der Hashtable bekannt ist, wird dieser vor allen anderen Zügen untersucht. Hier ist noch anzumerken, daß es natürlich keine Garantie gibt, daß dieser Zug tatsächlich der beste ist: Dies hängt von der verbleibenden Suchtiefe ab. Sollte diese für den gespeicherten Zug kleiner sein als diejenige, zu der beim erneuten Erreichen der Position gesucht wird, so könnte sich bei der tieferen Suche ein anderer Zug als beste Möglichkeit herausstellen.

Die **Killer Heuristic** ist ein weiteres Verfahren zur Vorsortierung, das ich verwenden werde. Es stellt für jede Suchtiefe lokale Informationen, den besten Zug dieser Ebene betreffend, zur Verfügung. Es ist relativ einfach und sehr effizient zu implementieren. Die Informationen der Killer

12 Außer im optimalen Fall, der aber in der Praxis nur äußerst selten auftritt

Heuristic stellen eine gute Ergänzung zur Hashtable-Vorsortierung dar, da die Züge, die in der Killer Heuristic gespeichert sind, meist aktuellere (weil lokalere) Informationen darstellen als Züge aus der Hashtable. Sollte diese lokale Information die gleiche sein wie die in der Hashtable, so ist sie redundant, hat jedoch keinen negativen Einfluß auf die Vorsortierung. Sollte sich aber bei der Wiederholung einer Suchtiefe ein Zug als bestmöglich herausstellen, der vorher nicht berücksichtigt wurde, so wird dieses Ergebnis sofort in der Killer Heuristic verwendet. Sollte an einer anderen Position als erstes ein „altes“ Ergebnis aus der Hashtable verwendet werden, so ist zumindest sichergestellt, daß der in der Killer Heuristic gespeicherte (tatsächlich beste) Zug als zweiter untersucht wird, also immer noch eine relativ gute Vorsortierung besteht. Die Stärke der Killer Heuristic besteht also darin, lokale Informationen zu sammeln und zur Verfügung zu stellen, die eine gute Ergänzung zu den globalen, oft über mehrere Suchen verwendeten Informationen der Hashtable darstellen. Die Killer Heuristic ist ein weiterer Ansatz, um die Anforderungen 2.4 und 2.6 zu erfüllen.

Die Verwendung eines **Iterative Deepening Frameworks** hat sich in der Schachprogrammierung ebenfalls als Standard durchgesetzt [Marsland 91], da sie die gleichen Anforderungen erfüllt. Durch die mehrfache Wiederholung der Suche mit steigender Suchtiefe entstehen verschiedene Möglichkeiten, ihren Ablauf zu beschleunigen: Da der Aufwand der Suche pro zusätzlicher Suchebene exponentiell wächst, können bei geringen Suchtiefen Informationen mit wenig Aufwand gesammelt werden, die später bei Suchen zu hohen Tiefen enorm effizienzsteigernd zum Einsatz kommen. Der größte Vorteil, im Sinne höherer Geschwindigkeit, entsteht durch eine bessere Vorsortierung an der Wurzel. Da, wie oben dargestellt, zu jeder Position der beste Folgezug gespeichert wird, findet dies auch an der Wurzel Anwendung. Es besteht natürlich die Möglichkeit, daß es sich, wie ebenfalls gezeigt, in einer weiteren Suche nicht mehr um den besten Zug handelt, die Vorsortierung also nicht optimal ist. Gleichzeitig ist es jedoch äußerst unwahrscheinlich, daß diese Vorsortierung einen negativen Einfluß hat, da die Kenntnis des besten Zuges an der Wurzel auf einer großen Menge untersuchter Stellungen basiert: Der Fall, daß sich ein bester Zug, bei Erhöhung der Suchtiefe um eins, nun als sehr schlecht herausstellt, tritt äußerst selten auf. Sollte ein Zug tatsächlich nicht mehr der Beste sein, so liegt seine Bewertung trotzdem meist immer noch deutlich über dem Durchschnitt, sein Einfluß auf die Effizienz der Suche ist also positiv. Außerdem gibt es die Möglichkeit, den besten Zug in der Killer Heuristic zu finden und so die Vorsortierung an der Wurzel zu optimieren. Die Killer Heuristic profitiert ebenfalls von der Verwendung des Iterative Deepening - Verfahrens, da Informationen bestimmter Suchtiefen durch die iterative Wiederholung der Suche beständig erneuert und wiederverwendet werden können. Die Effizienzsteigerung durch Iterative Deepening wird also wiederum deutlich von der Verwendung der beiden oben genannten Verfahren beeinflußt, es entsteht ein Synergieeffekt. Basierend auf den gleichen Annahmen werde ich auch das Internal Iterative Deepening genannte Verfahren verwenden (s. 2.2.6.).

Ein Verfahren, das nicht zum Einsatz kommen wird, ist die **History Heuristic**. Dies geschieht aus zwei verschiedenen Gründen: Erstens sind die Informationen der History Heuristic zu global; sie werden über den gesamten Suchbaum gesammelt und ohne Berücksichtigung der Suchebene, der Stellung oder der bewegten Figur gespeichert. Die History Heuristic kann dadurch auch einen deutlich negativen Einfluß auf die Vorsortierung haben. Es scheint nicht sinnvoll, einen Zug als guten Kandidaten zu bewerten, nur weil er an einer beliebigen Stellung irgendwo im Suchbaum ein bester Zug war: So kann es z.B. sinnvoll sein, durch den Zug b3f5 einen Läufer zum Tausch gegen einen anderen Läufer anzubieten, der gleiche Zug mit der Dame wäre jedoch eine sehr schlechte Wahl. Die History Heuristic ist nur dann effizienzsteigernd, wenn keine anderen Verfahren zur Vorsortierung zur Verfügung stehen. Da ich mich aber zur Verwendung der beiden weiter oben genannten Verfahren entschieden habe, wären die Informationen der History Heuristic zum großen Teil redundant, wenn nicht sogar kontraproduktiv.

Der zweite Grund die History Heuristic nicht zu verwenden, ist implementationsspezifisch: Da die von mir antizipierte Hardwareumgebung sich nicht durch hohe Rechengeschwindigkeit auszeichnet, sollte man, zur Erfüllung der Anforderung 2.4, jeden numerischen Aufwand vermeiden. Ein Teil dieses Aufwands ist die Sortierung der möglichen Züge. Da für jeden Knoten im Suchbaum diese Sortierung notwendig ist, ist dieser Aufwand ein Teil des globalen exponentiellen Aufwands, den es zu reduzieren gilt. Deshalb werde ich die Sortierung derart implementieren, daß nicht alle Züge zu sortieren sind (was mindestens dem lokalen Aufwand O(n * log(n)) entsprechen würde), sondern nur eine Untermenge aussichtsreicher Kandidaten. Dies ist mit deutlich weniger Aufwand als einer kompletten Sortierung zu bewerkstelligen, was dem Mangel an Rechengeschwindigkeit der Zielplattform entgegenkommt. Sollte nun die History Heuristic verwendet werden, so enthält diese durch ihren globalen Informationsgewinn bereits nach kürzester Zeit für jeden möglichen Zug einen History-Wert, der angibt, inwieweit der Zug ein aussichtsreicher Kandidat ist, also nach vorne sortiert werden sollte. Da es keine logische Obergrenze für den möglichen History-Wert eines Zuges gibt, wäre eine partielle Sortierung nicht mehr sinnvoll zu implementieren; auf Basis der History-Werte eine vollständige Sortierung durchzuführen, stünde aber vom rechnerischen Aufwand her in keinem Verhältnis zu dem zu erwartenden Performanzgewinn.

Static Exchange Evaluation ist ein Verfahren, das ich ebenfalls nicht verwenden werde. Die Begründung hierfür ist wiederum der zu erwartende rechnerische Aufwand: SEE kann einerseits in einer rekursiven Variante implementiert werden, dies ist für eine performanzorientierte Anwendung jedoch auszuschließen, da der Aufwand einer zusätzlichen rekursiven Funktion, die ausschließlich zu Mitteln der Vorsortierung benutzt wird, nicht zu rechtfertigen ist. Es gibt auch die Möglichkeit einer nicht-rekursiven Variante, die aber ebenfalls erheblichen rechnerischen Aufwand erzeugt. SEE bringt nur dann einen Vorteil, wenn die Zielplattform ein effizientes Handling von 64-Bit Werten ermöglicht. Die Schlagzüge einer Figur werden nämlich als 64-Bit Werte dargestellt, wobei jedes Bit ein Feld des Schachbretts repräsentiert; ein einzelner Wert kann also alle Schlagzüge beschreiben, die für eine bestimmte Figur möglich sind. Um nun die für eine Static Exchange Evaluation notwendigen Berechnungen anzustellen, ist es notwendig, diese 64-Bit Werte in einer hohen Geschwindigkeit verarbeiten zu können, dies ist auf dem HP iPAQ H5500 jedoch nicht möglich.
Ich habe also, ähnlich wie oben, entschieden, daß den Begrenzungen der Rechenkapazität der Zielplattform in der Art Rechnung getragen werden muß, Verfahren auszuschließen, die einen zu hohen und zur Durchführung der Suche nicht zwingend notwendigen Aufwand darstellen. Dies dient der Erfüllung der Anforderungen 2.4 und 2.6.

4.3.3. Weitere Verfahren

Zur Steigerung der Geschwindigkeit der Suche werde ich außerdem **Verified Nullmove Pruning** verwenden. Nullmove Pruning allgemein ist für moderne Schachprogramme unverzichtbar, da es, wie in 2.2.8. erläutert, sehr starke Geschwindigkeitsverbesserungen ermöglicht. Um die möglichen Nachteile dieses Verfahrens in Grenzen zu halten, habe ich mich für das Verified Nullmove Pruning entschieden, daß zwar u.U. nur geringere Geschwindigkeitsverbesserungen ermöglicht als das einfache Nullmove Pruning, dafür aber wesentlich sicherere Ergebnisse liefert. Zur Erfüllung der Anforderung 2.3 nach korrekten Suchergebnissen ist dies unbedingt zu berücksichtigen.

Um die Effizienz der Suche zu erhöhen, werde ich außerdem **Futility Pruning** sowie **Lazy Evaluation** implementieren. Diese beiden Verfahren sind sehr gut geeignet, um die Menge der zu untersuchenden Knoten zu reduzieren und dadurch die Suche zu beschleunigen. Je nach Art der praktischen Umsetzung können diese Verfahren so implementiert werden, daß keine Knoten

abgeschnitten werden, die einen Einfluß auf das Suchergebnis haben können. Diesen Ansatz habe ich für meine Implementation genutzt, um, ähnlich wie beim Verified Nullmove Pruning, die Korrektheit der Ergebnisse garantieren zu können, also die Anforderung 2.3 zu erfüllen. Aus diesem Grund werde ich sowohl für Futility Pruning als auch für Lazy Evaluation einen theoretischen Maximalwert der Evaluation als Schnittwert verwenden, der dem tatsächlichen Maximalwert entspricht (s. 2.2.9. und 2.2.10.). Auf diese Art haben die beiden Verfahren keinen Einfluß auf die Genauigkeit des Suchergebnisses, die erreichbaren Geschwindigkeitsvorteile aber fallen bei einer Implementation in einer Umgebung mit stark begrenzten Ressourcen noch deutlicher ins Gewicht, was hilft, die Anforderung 2.4 zu erfüllen.

4.3.4. Anmerkungen

Im Bereich der Schachprogrammierung stellt sich dem Entwickler oft die Frage, welche Verfahren aus der großen Menge der möglichen Ansätze auszuwählen sind. Dies wird zum Teil durch die Kapazitäten der Zielplattform bestimmt: Die Einschränkungen der in meiner Arbeit betrachteten Geräte liegen vor allem in der mangelnden Größe des Speichers sowie der niedrigen Taktfrequenz der CPU. Deshalb versuche ich für eine Anwendung in diesen Systemen auf Verfahren zu verzichten, die entweder, wie MTD(f), zuviel Speicher benötigen, oder aber einen großen, vermeidbaren rechnerischen Aufwand mit sich bringen - wie SEE und History Heuristic.

Zum anderen gibt es deutliche **Wechselwirkungen** zwischen den verwendeten Verfahren: Wie bereits bei den Verfahren zur Vorsortierung dargestellt, sind Hashtable-Vorsortierung, Killer Heuristic und History Heuristic zum Teil redundant. Da zu ihrer Anwendung ein gewisser rechnerischer Aufwand nötig ist, ist es durchaus möglich, alle diese Verfahren zu verwenden und trotzdem eine niedrigere Effizienz zu erreichen, als wenn man nur ausgewählte Ansätze verwenden würde.
Ebenso kann es sein, daß bestimmte Verfahren in einer kombinierten Anwendung erst ihre wahren Möglichkeiten entfalten, dies gilt vor allem für die Verwendung eines Iterative Deepening Frameworks, das erstaunliche Performanzgewinne ermöglicht. Dies geschieht vor allem dadurch, daß Hashtables und Killer Heuristic sehr stark von den Wiederholungen der Suche profitieren, und auf diesem Wege eine deutlich höhere Effizienz erreichen.
Diese beiden Beispiele zeigen die Problematik, die sich bei der Anwendung verschiedener Optimierungsmaßnahmen ergibt: Mögliche Wechselwirkungen müssen ständig beobachtet werden, um eine wirkliche Optimierung des Ablaufs zu erreichen. Es ist also durchaus nicht sinnvoll, alle möglichen Verfahren zu implementieren, vielmehr muß eine Auswahl getroffen werden, die ein möglichst gutes Zusammenspiel der verwendeten Ansätze ermöglicht.

Ein weiterer Faktor, der berücksichtigt werden muß, sind mögliche **Seiteneffekte**. So muß z.B. bei Verwendung des Nullmove Pruning berücksichtigt werden, daß Züge innerhalb einer Nullmove-Suche nicht für die Killer Heuristic verwendet werden dürfen. Die Killer Heuristic geht nämlich davon aus, daß bei einer bestimmten Suchtiefe immer der gleiche Spieler am Zug ist. In einer Nullmove-Suche ist es aber so, daß ein Spieler zweimal hintereinander ziehen darf, das Zugrecht also für die folgenden Suchtiefen vertauscht wird. Sollten jetzt Züge in der Killer Heuristic gespeichert werden, so wären sie für den normalen Suchablauf nutzlos, da sie nur in einer Nullmove-Suche erzeugt werden können, auf diese Weise würden sie die Effizienz der Killer Heuristic vermindern.
Solche Seiteneffekte entstehen in großer Zahl, deshalb ist es wichtig, vor der Implementation eines Verfahrens genau zu analysieren, welche Veränderungen für den gesamten Ablauf sich daraus ergeben. Sind die möglichen negativen Auswirkungen zu groß, so muß das Verfahren modifiziert oder aber von seiner Verwendung gänzlich abgesehen werden.

In einer abstrakten Betrachtung stellen viele Verfahren einen Tausch „Wissen gegen Geschwindigkeit“ dar. So dienen sie dazu, die Suche mit zusätzlichen Informationen zu versorgen, um einen schnelleren Ablauf zu ermöglichen; trotz der sehr unterschiedlichen Ansätze ist dies das eigentliche Ziel aller Optimierungsmaßnahmen, die zu einer **informierten Suche** führen. Gleichzeitig aber verringern diese Verfahren die Ausführungsgeschwindigkeit, da zusätzlicher rechnerischer Aufwand geleistet werden muß. Es liegt nun am Entwickler, das Verhältnis dieser beiden Elemente richtig einzuschätzen und daraus abzuleiten, welche Verfahren lohnenswert sind und welche keinen Gewinn versprechen. Dieses Feld ist offen für viele verschiedene Ansätze und Optimierungsmöglichkeiten, von denen ich einige vorgestellt habe.

4.4. Implementation der MinMax-Suche

4.4.1. Die Knoten-Struktur

Für die tatsächliche Implementation einer MinMax-Suche bedarf es einer internen Repräsentation der Knoten, aus denen der Suchbaum aufgebaut wird. Da ich mich bemüht habe, die Suche so performant wie möglich zu implementieren, habe ich auf Elemente der Objektorientierung verzichtet, ein Knoten wird in meiner Anwendung durch ein einfache C-Struktur dargestellt:

```
typedef struct {
   unsigned int key;
   unsigned int lock;
   short int board[64];
   short int material[2];
   unsigned short int kpos[2];
   unsigned short int phase;
   unsigned char check;
   unsigned char extension;
   unsigned char fifty;
   char enpassant;
} Node;[13]
```

Ich werde nun die einzelnen Elemente dieser Struktur erläutern:
Die beiden Variablen `key` und `lock` enthalten den Zobrist-Schlüssel der Stellung, werden dabei aber aus zwei verschiedenen Mengen an Teilschlüsseln aufgebaut. Sie werden im Ablauf der Suche benutzt, um den Hashtable-Zugriff zu ermöglichen sowie Schlüsselkollisionen bei diesem Zugriff zu erkennen. Außerdem sind sie notwendig, um Zyklen im Suchgraphen zu erkennen.
Das Spielfeld wird durch das Array `board` repräsentiert, das für jedes Feld des Spielbretts einen direkten Zugriff ermöglicht. Ist ein Feld des Spielbretts nicht besetzt, so ist der Wert im `board` Array 0, für jede Figur wird der Wert des `board` Arrays auf den Materialwert der jeweiligen Figur gesetzt[14] (s. 4.5.5.), wobei weiße Figuren positive Werte erhalten, schwarze Figuren negative.
Das Array `material` enthält das Gesamtmaterial der beiden Spieler, dies wird ebenfalls durch positive Werte für Weiß und negative Werte für Schwarz dargestellt.
Um die Schach-Erkennung sowie andere Funktionen zu vereinfachen, werden die Positionen der beiden Könige in einem eigenen Array gespeichert, `kpos` enthält diese Positionen.
Da ich eine Evaluations-Funktion entwickelt habe, die Stellungen abhängig von der jeweiligen Phase des Spiels bewertet (abgeleitet von den möglichen Spielzuständen Eröffnung, Mittelspiel und Endspiel, s. 4.5.5.), ist es notwendig, diese Phase als Element eines Knotens zu verwalten, sie wird durch den Wert `phase` repräsentiert.
Die Variable `check` gibt an, ob der aktive Spieler an diesem Knoten im Schach steht. Dies muß unbedingt bekannt sein, um Suchergebnisse korrekt bewerten zu können; außerdem sind viele Optimierungsverfahren dann nicht anwendbar.
Da, wie in 2.1.9. dargestellt, durch Search Extensions ein selektiver Anteil in die Suche integriert wird, ist es notwendig, die mögliche Vertiefung der Suche an einem Knoten darzustellen, hierzu wird die Variable `extension` verwendet.
Das Element `fifty` ist ein Zähler, der angibt, wieviel Halbzüge seit dem letzten Schlagzug oder dem letzten Zug mit einem Bauern vergangen sind. Es ist notwendig, diesen Wert zu kennen, um die 50-Zug-Regel zu implementieren (s. Anhang D „FIDE-Schachregeln“).
Die Variable `enpassant` erhält standardmäßig den Wert -1. Sie wird nur dann gesetzt, wenn an dem betreffenden Knoten ein En-Passant-Zug möglich ist, in diesem Fall erhält sie den Index des Feldes, auf das dieser Zug ausgeführt werden kann.

13 In Kapitel 4 werden Auszüge aus dem Quellcode der Anwendung zur Visualisierung verwendet
14 Dieser Ansatz schließt aus, daß Figuren Identität besitzen

Die hier dargestellte Repräsentation eines Knotens enthält ausschließlich die für einen korrekten Ablauf der MinMax-Suche absolut notwendigen Elemente. Aus Rücksicht auf die starken Limitierungen der Zielplattform habe ich darauf verzichtet, Elemente zu verwenden, die strukturelle Vereinfachungen des Programmablaufs ermöglichen würden: Hier ist vor allem die Möglichkeit zu nennen, Figuren und ihre Positionen in einer eigenen Liste darzustellen. Die Daten dieser Liste wären mit den Daten im `board` Element redundant, würden jedoch einen verständlicheren Ablauf verschiedener Funktionen, wie z.B. der Zugerzeugung oder der Bewertungsfunktion, erlauben.

Denkbar wäre auch, sog. Bitboards zu verwenden. Bitboards repräsentieren alle möglichen Züge einer Figur in einem 64-Bit Wert. Mit ihrer Hilfe kann man verschiedene Funktionen, wie z.B. die Schacherkennung, deutlich vereinfachen. Da alle Schlagzüge einer Figur in einem einzigen Wert bekannt sind kann man durch logische Verknüpfung dieser Werte leicht erkennen, ob der König im Schach steht. Bitboards können ebenfalls verwendet werden, um eine SEE zu implementieren, ich habe jedoch aus den in 4.3.2. bereits dargestellten Gründen darauf verzichtet.

4.4.2. Suchfunktionen

Die MinMax-Suche besteht aus vier Funktionen, die im Ablauf einer Suche erst sukzessiv, dann rekursiv zum Einsatz kommen. An der Wurzel wird die `StartNegaScoutIterative` - Funktion aufgerufen, die Iterative-Deepening unter Anwendung eines Aspiration Window implementiert (s.u.). Hier werden die Ergebnisse der eigentlichen Suche verarbeitet und an CEBoard weitergegeben. Außerdem übernimmt diese Funktion die Aufgabe der **Zeitkontrolle**, also der Einteilung der verfügbaren Zeit für verschiedene Suchen: Sollte die für einen weiteren Durchlauf des Iterative Deepening noch zur Verfügung stehende Zeit kleiner als ein Drittel der anfänglichen Zeit sein, so wird die Suche abgebrochen, da angenommen werden kann, daß innerhalb dieser Zeit keine neue Suchtiefe bewältigt werden kann. Sollte sich aus einem Suchergebnis ein Aspiration Window Fail Low ergeben, so wird die mögliche Zeit für eine Wiederholungssuche verdoppelt (s. 2.1.3.).
Die Implementation des PVS/NegaScout-Algorithmus ist in zwei verschiedene Funktionen unterteilt: Die erste Funktion `StartNegaScout` wird für die oberste Ebene der Suche aufgerufen; sie unterscheidet sich von der Funktion für den Rest der Suche, da hier der beste Zug gefunden werden muß, nicht nur das beste Ergebnis. Außerdem wird sie pro Suche nur einmal verwendet, ist also bei weitem nicht so zeitkritisch, wie die `NegaScout` - Funktion, die über einen rekursiven Aufruf die eigentliche Tiefensuche realisiert.
Zu Beginn der `StartNegaScout` - Funktion wird die Zugliste erzeugt, um zu prüfen, ob nur ein legaler Zug zur Verfügung steht. Wenn ja, wird dieser sofort zurückgegeben, und die Zeit für eine mögliche Suche kann eingespart werden. Als nächstes erfolgt der Transposition Table - Zugriff, wobei Einträge der Eröffnungsbibliothek ebenfalls sofort zurückgegeben werden. Sollte ein sonstiger bester Zug gefunden werden, so wird er als erstes untersucht. Auf der obersten Ebene der Suche erfolgt keine Anpassung der Alpha- und Beta-Werte durch ein mögliches Transposition Table - Ergebnis, um hier eine größere Menge exakter Ergebnisse zu erhalten.
Nach dem Ausspielen jedes Zuges werden das aktuelle Suchergebnis sowie andere Daten der Suche an CEBoard gesendet. Sollte ein Ergebnis eine deutliche Verschlechterung (entsprechend der Größe des halben Aspiration Window) gegenüber der letzten Suche bedeuten, so wird die Zeit für diese Suche verdoppelt, jedoch nur, wenn sie nicht bereits durch einen Aspiration Window Fail erhöht wurde. An der Wurzel gibt es außerdem eine zweite Zeitkontrolle: Wenn bereits 30% der möglichen Züge untersucht wurden, so wird die Maximalzeit um 30% erhöht, allerdings nur, wenn die Zeit für die aktuelle Suche nicht bereits anderweitig erhöht wurde. Dies soll die Annahme wiederspiegeln, daß es effizienter ist, eine Suche zu Ende zu führen, wenn man bereits einen

signifikanten Anteil der Züge untersucht hat, als die Suche an dieser Stelle abzubrechen. Sollte der Anteil der bereits untersuchten Züge bei über 60% liegen, so wird die Zeit pro weiterem Zug um 30% erhöht, jedoch nur solange, bis ein Maximalwert, entsprechend dem vierfachen Wert der ursprünglichen Suchzeit, erreicht wird.
Abgesehen von den hier genannten Eigenschaften sind die Unterschiede der `StartNegaScout` - Funktion und der `NegaScout` – Funktion der Abbildung 4.5 (s.u.) zu entnehmen.
Wenn innerhalb der `NegaScout` - Funktion der Suchhorizont erreicht wird, so muß die Quiescence Search gestartet werden. Dieser Teil der Suche ist in der `AlphaBetaQ` – Funktion implementiert, die ebenfalls rekursiv abläuft. Den Ablauf dieser **rekursiven Suchfunktionen** werde ich im nächsten Abschnitt weiter beschreiben.

4.4.3. Ablauf einer Suche[15]

Zu Beginn dieses Abschnitts werden die in der MinMax-Suche implementierten Verfahren anhand einer Grafik (s.u.) zusammenfassend vorgestellt. Jeder Suchfunktion werden dabei die verwendeten Verfahren zugeordnet, wobei diese in der Reihenfolge dargestellt werden, wie sie im Ablauf der betreffenden Funktion zum Einsatz kommen. Außerdem wird durch einen Suchbaum gezeigt, welche Funktion in welchem Stadium der Suche verwendet wird.
Im weiteren Verlauf des Abschnitts folgt ein Beschreibung des Suchablaufs, wobei ich mich auf die rekursiven Suchfunktionen konzentrieren werde. Die Implementation von Verfahren, die nicht bereits in anderen Kapiteln erläutert wurde, wird hier durch Code-Fragmente dargestellt. Hierbei wird zusätzlich gezeigt, wo diese Verfahren in der MinMax-Suche implementiert wurden und welche Details es bei der Implementation zu beachten gilt.

15 Im Folgenden wird die Abkürzung TT für Transposition Table verwendet

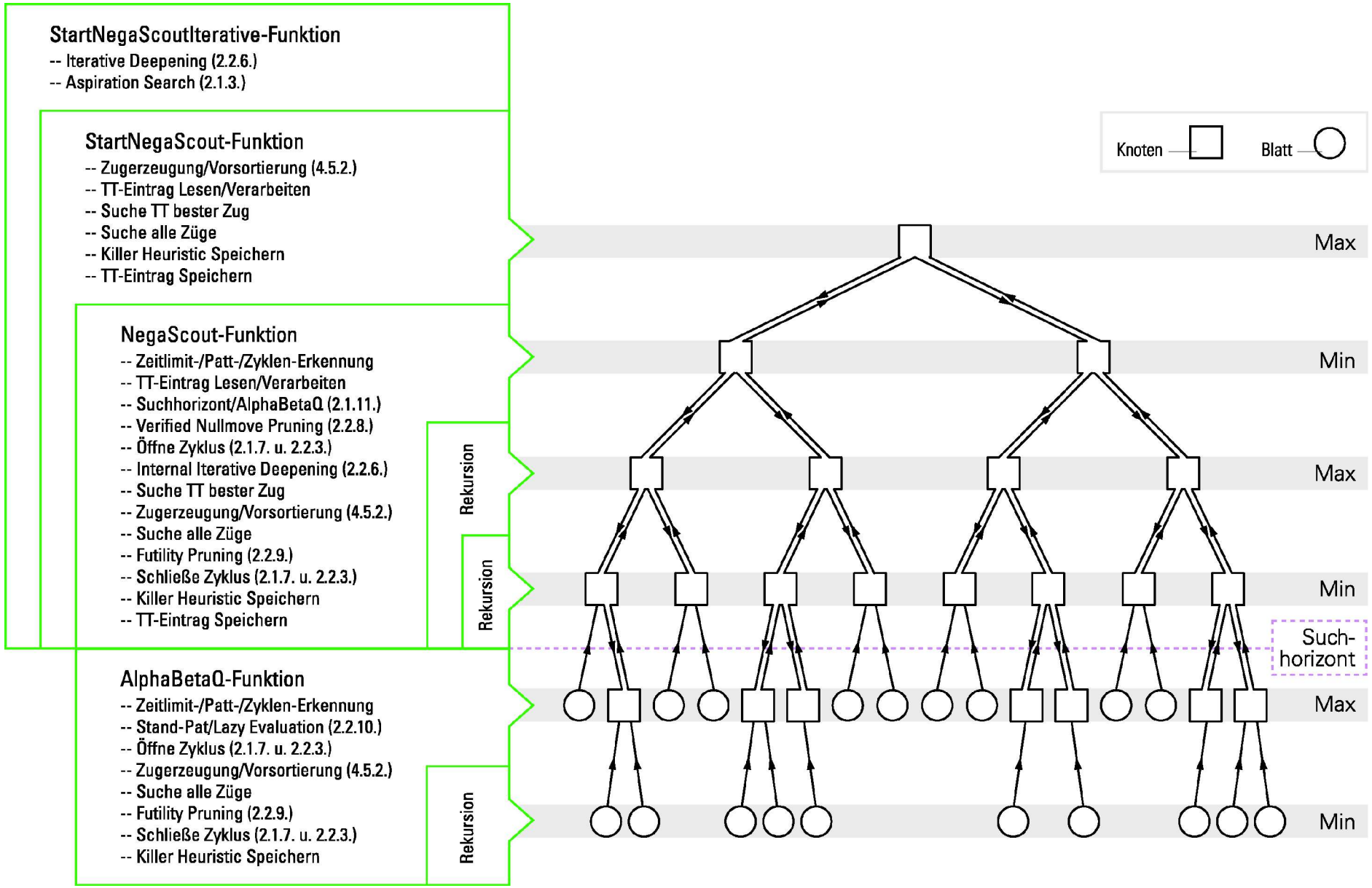


Abb. 4.5: Ablauf einer Suche anhand der Suchfunktionen und der dort angewendeten Verfahren

Den Beginn einer Suche stellt immer das **Iterative Deepening Framework** dar. Von hier wird mit sukzessive steigender Suchtiefe die eigentliche Suchfunktion aufgerufen. Der Übergabeparameter `char player` (s.u.) beschreibt den an der Wurzel aktiven Spieler: `WHITE = 1`, `BLACK = -1`.
Vor Beginn der ersten Suchtiefe müssen zunächst die Datenstrukturen verschiedener Heuristiken aufgearbeitet werden: So ist es notwendig, die Daten der Killer Heuristic zu löschen, da sie Züge der letzten Suche enthält, also Ergebnisse, die, bezogen auf ihre Suchtiefe, nicht mehr gültig sind. Außerdem verwende ich Hashtables mit einem Alterungsfaktor (s. 4.5.3.), um die Verstopfung mit alten Daten zu verhindern, dieser Faktor muß hier ebenfalls manipuliert werden.
Als nächster Schritt erfolgt die Anwendung des **Aspiration Windows** für die folgende Suche. Beim ersten Durchlauf (in der Praxis meist zur Tiefe zwei, also mit vernachlässigbaren Kosten verbunden) setzt man dieses Fenster auf minus unendlich und plus unendlich, da noch keine Informationen über das zu erwartende Ergebnis vorliegen. Ist eine Suchtiefe beendet, wird man das Suchfenster für den nächsten Durchlauf in einem Bereich um das Ergebnis dieser Suche anordnen.
Außerdem wird die durch Extensions maximal erreichbare Tiefe einer Suche `int depth_max` bestimmt, die später benötigt wird, um den Suchhorizont zu erkennen.

```
int StartNegaScoutIterative(Node n, char player) {

   RebuildTables();              //Aufarbeitung der Datenstrukturen

   int epsilon = PAWN / 3;       //Größe des Aspiration-Windows

   int alpha = -INT_INF;         //offenes Suchfenster
   int beta = +INT_INF;

   char verify = TRUE;           //benötigt für Verified Nullmove Pruning (s.u.)

   for (int depth = MINDEPTH; depth <= MAXDEPTH; depth++) {

      int depth_max = (((depth - 1) + ((depth / 2) + 1)) - 1); //Maximaltiefe

      int merit = StartNegaScout(n, depth, alpha, beta, verify, player);

                                                    //Wiederholungssuche?
      if (merit >= beta)
         merit =  StartNegaScout(n, depth, merit, +INT_INF, verify, player);

      else if (merit <= alpha)
         merit =  StartNegaScout(n, depth, -INT_INF, merit, verify, player);

      alpha = merit - epsilon;                      //begrenztes Suchfenster
      beta = merit + epsilon;
   }

   return merit;
}
```

Nach dem Aufruf der `StartNegaScout` – Funktion erfolgt die rekursive Tiefensuche, die in der `NegaScout` - Funktion implementiert ist. Zu Beginn dieser Funktion kann abgebrochen werden, wenn das gegebenen Zeitlimit überschritten wurde[16]. Wenn dies nicht der Fall ist, prüft man, ob es sich um einen Zyklus im Suchgraphen handelt (s. 2.1.7. und 2.2.3.). Wenn ja, kann die Suche ebenfalls beendet werden. Desweiteren wird geprüft, ob es sich um ein Patt aufgrund der 50-Zug-Regel handelt. Außerdem wird getestet, ob ein Patt aufgrund mangelnden Materials vorliegt: Sollten beide Spieler keine Figuren außer der des Königs besitzen, so handelt es sich um ein Patt, da nicht mehr mattgesetzt werden kann. Dies gilt auch für den Fall, daß beide Spieler keine Figuren außer einer oder zwei Leichtfiguren (Springer oder Läufer) besitzen, weil mit diesen Figuren ein

16 Dies wird außerdem in einem globalen Flag markiert, um die Speicherung irregulärer Suchergebnisse zu verhindern

Mattsetzen ebenfalls nicht möglich ist. Eine Ausnahme hiervon ist das Läuferpaar, hiermit ist es möglich, ein Matt zu erreichen[17]. Sollte ein Patt erkannt werden, wird der Wert null zurückgegeben, es wird also kein contempt factor (s. 2.1.10.) verwendet.

```
//Time Cut
if (GetTimeInterval(stime) > stime_max) return TIMECUT;

//Draw by Repetition
if (Repetition(&(n.key), &(n.lock))) return DRAW;

//Draw by 50 Move
if (n.fifty == 100) return DRAW;

//Draw by Material
if (DRAWMATERIAL(n.material[0]) && DRAWMATERIAL(n.material[1])) return DRAW;
```

Nun wird geprüft, ob die lokale Suchtiefe durch mögliche Search Extensions (s. 4.5.4.) zu vergrößern ist, wenn ja, findet dies hier statt:

```
//Extension
if (n->extension >= 4) {
   depth += (n->extension / 4);
   n->extension = 0;
}
```

Als nächstes erfolgt der Zugriff auf die **Transposition Table.** Die Ergebnisse werden dabei für Alpha- oder Beta-Schnitte verwendet. Die möglichen Werte des `tt_flag` sind `LBOUND` (Untergrenze), `UBOUND` (Obergrenze) und `VALID` (exaktes Ergebnis)[18]. Über den Wert `MATE_THRESHOLD` (= -10000) wird geprüft, ob ein Schachmatt Ergebnis gespeichert wurde (s.u.). Ist es nicht möglich, einen Schnitt zu erzeugen, so wird versucht, das Suchfenster mit Hilfe des Transposition Table-Ergebnisses zu verkleinern:

```
//TT Retrieve
short int tt_merit;
char tt_flag;
char tt_depth = -CHAR_INF;
unsigned short int tt_move = NOMOVE;
                                                //Stellung gespeichert?
if (Retrieve(&(n.key), &(n.lock), &tt_depth, &tt_merit, &tt_flag, _
   &tt_move)) {

   //Mating or Mated?
   if (tt_merit > -MATE_THRESHOLD) {
      if (tt_flag == VALID || tt_flag == LBOUND) return tt_merit;
   } else if (tt_merit < MATE_THRESHOLD) {
      if (tt_flag == VALID || tt_flag == UBOUND) return tt_merit;
   }

   //Enough Information?
   if (tt_depth >= depth) {

      //Cutoff?
      if (tt_flag == VALID) return tt_merit;
      if (tt_flag == LBOUND && tt_merit >= beta) return tt_merit;
      if (tt_flag == UBOUND && tt_merit <= alpha) return tt_merit;
```

17 Dies gilt theoretisch auch für die Kombination Springer und Läufer, ist aber nur mit enormem rechnerischen Aufwand zu bewerkstelligen

18 Es gibt noch den Wert `BOOK`, der aber hier nicht relevant ist (s. 4.5.1.)

```
      //Adjust Bounds
      if (tt_flag == LBOUND || tt_flag == VALID)
         alpha = MAX(alpha, tt_merit);
      if (tt_flag == UBOUND)
         beta = MIN(beta, tt_merit);
   }
}
```

Im weiteren Ablauf der Suche muß nun geprüft werden, ob der Suchhorizont erreicht wurde. Der Wert `depth` beschreibt hierbei die verbleibende Suchtiefe, der Wert `d` die Suchtiefe, die bereits von der Wurzel aus zurückgelegt wurde. Diese doppelte Prüfung ist notwendig, um die Gesamtmenge der oben dargestellten Extensions limitieren zu können. Ist der Suchhorizont erreicht, wird die **Quiescence Search** aufgerufen und ihr Ergebnis zurückgegeben:

```
//Horizon?
if (depth <= 0 || d > depth_max)
   return AlphaBetaQ(n, MAXDEPTHQ, alpha, beta, player);
```

Die Quiescence Search wurde in Nemesis als einfache Alpha/Beta-Suche implementiert. Dies ist sinnvoll, da sie mit den aktuellen Alpha- und Beta-Werten aufgerufen wird, die aufgrund des aufrufenden PVS/NegaScout-Algorithmus in fast allen Fällen bereits ein geschlossenes Suchfenster beschreiben. Die Verwendung einer Nullfenstersuche innerhalb der Quiescence Search würde also nur zusätzlichen Aufwand erzeugen.
Die Transposition Table wird in dieser Funktion nicht verwendet, da die Quiescence Search für jeden Zyklus des Iterative Deepening ab einer neuen Tiefe, dem Horizont der MinMax-Suche, gestartet wird. Dies schränkt die Möglichkeit der Wiederverwendung von Ergebnissen deutlich ein, da es (auch aufgrund der verwendeten Vereinfachungen) kaum zu Wiederholungen von Knoten kommt.
Zu Beginn der `AlphaBetaQ` - Funktion wird auf verschiedene Möglichkeiten geprüft, ein Patt zu erzeugen (s.o.), sollte dies der Fall sein, so wird das Ergebnis null zurückgegeben. Für die Quiescence Search wird ebenfalls eine Maximaltiefe verwendet, um den rechnerischen Aufwand in den Grenzen zu halten, die für die Zielplattform akzeptabel sind. Sollte diese Maximaltiefe erreicht werden, so wird das Ergebnis der statischen Evaluation zurückgegeben (außer der aktive Spieler steht im Schach).
Die in Nemesis implementierte Quiescence Search verwendet die Möglichkeit eines Stand-Pat – Beta-Schnitt, wobei Lazy Evaluation zum Einsatz kommt: Sollte der Wert der Lazy Evaluation ausreichend sein für den Stand-Pat – Beta-Schnitt, so wird er zurückgegeben, ist dies nicht der Fall, so wird der tatsächliche Evaluationswert berechnet. Ist dieser Wert größer Beta, so kann abgebrochen werden, ansonsten wird Alpha angepasst:

```
//Horizon And Stand-Pat Beta Cutoff
if (!n.check) {

   //Horizon?
   if (depth <= 0) return player * Eval(n);

   //Lazy Eval
   merit = (player * (n.material[0] + n.material[1])) - MAXEVAL;
   if (merit >= beta) return merit;

   //Full Eval
   merit = player * Eval(n);
   if (merit >= beta) return merit;

   alpha = MAX(alpha, merit);
}
```

Danach wird die Zugliste erzeugt, wobei nur Schlagzüge berücksichtigt werden; sollte der aktive Spieler allerdings im Schach stehen, so werden alle möglichen Züge erzeugt, um Schachmatt erkennen zu können. Nun werden die Kindknoten untersucht, innerhalb dieser Suche wird Futility Pruning (s.u.) verwendet; ich benutze das Verfahren dabei in der Art, daß ich ausschließlich den tatsächlichen Maximalwert der Evaluation für mögliche Schnitte verwende. Dies erlaubt zwar etwas weniger Schnitte, stellt aber gleichzeitig die maximale Qualität der Ergebnisse sicher.
Ist die Untersuchung der Kindknoten beendet, so wird geprüft, ob der beste Zug für die Killer Heuristic verwendet werden kann. Dies ist dann der Fall, wenn kein Alpha-Fail vorliegt und die Quiescence Search nicht von einer vorherigen Nullmove-Suche aufgerufen wurde (s. 4.3.4.). Danach wird das Ergebnis zurückgegeben.

Sollte der Suchhorizont nicht erreicht worden sein, so muß im Ablauf der `NegaScout` - Funktion als nächstes geprüft werden, ob **Verified Nullmove Pruning** angewendet werden kann. Dazu gibt es verschiedene Vorbedingungen: Als erstes muß sichergestellt werden, daß der aktive Spieler nicht im Schach steht. Wäre dies der Fall, würde die Nullmove-Suche zu irregulären Ergebnissen führen, da der andere Spieler den König des aktiven Spielers schlagen könnte, der Nullmove also eine illegale Stellung erzeugen würde (s. 2.1.6.). Desweiteren wird geprüft, ob eine Verifikation des Nullmove-Ergebnisses notwendig ist, und wenn ja, wird Verified Nullmove Pruning nur ausgeführt, wenn diese nach der Tiefenreduzierung noch möglich ist:

```
//Verified Nullmove Pruning
char fail_high = FALSE;
char nmdepth = depth;

if (!n.check && (depth > 1 || !verify)) {         //Vorbedingungen
   short int r = 3;
   short int nmd = (depth - r - 1);               //Tiefe der Nullmove-Suche
   unsigned int _key = n.key;
   unsigned int _lock = n.lock;
   char _enpassant = n.enpassant;
   ZobristNullmove(&(n.key), &(n.lock));
   n.enpassant = -1;
                                                  //TT Annahme (s.u.)
   if (tt_depth < nmd || tt_merit > beta || tt_flag == LBOUND) {

                                                  //Nullmove-Suche
      short int nms = -NegaScout(n, nmd, -beta, -beta+1, verify, -player);

      if (nms >= beta) {                          //möglicher Beta-Schnitt

         if (verify && nms < -MATE_THRESHOLD) {   //Verifikation notwendig?
            depth -= (depth > 3) ? 2 : 1;         //Tiefenreduktion für die
                                                  //tatsächliche Suche
            verify = FALSE;                       //Verifikation nicht mehr
                                                  //notwendig
            fail_high = TRUE;                     //Markierung eines
                                                  //möglichen Beta-Schnitts
         } else return nms;                       //Beta-Schnitt
      }

      //Mate Threat Extension (s. 4.5.4.)
      if (nms < MATE_THRESHOLD) n.extension += 3;

      n.key = _key;
      n.lock = _lock;
      n.enpassant = _enpassant;
   }
}
```

Es gibt einen möglichen Ansatz, um die Effizienz dieses Verfahrens zu steigern: Wenn ein Ergebnis aus der Transposition Table für die aktuelle Position bekannt ist, so wird Verified Nullmove Pruning nur ausgeführt, wenn dieses Ergebnis aus einer kleineren Tiefe kommt als die Tiefe der Nullmove-Suche (also nicht genügend Informationen enthält), oder aber sichergestellt ist, daß das Ergebnis weder ein exakter Wert noch eine Obergrenze kleiner Beta ist. Man geht hierbei davon aus, daß, wenn ein Ergebnis kleiner Beta bekannt ist, es als sehr unwahrscheinlich gelten kann, daß eine Nullmove-Suche (also eine Suche, bei der der Gegner zweimal ziehen darf) einen Wert größer Beta erreichen wird. In diesem Fall wird Verified Nullmove Pruning nicht angewendet. Es wird deshalb nur auf größer (nicht gleich) Beta geprüft, da für den Fall einer in der Transposition Table gefundenen Obergrenze Beta unter Umständen auf diesen Wert angepasst wurde, der Vergleich also für diesen Fall immer erfolgreich wäre. Da es sich aber um eine Obergrenze handelt, ist es ebenfalls nicht sinnvoll anzunehmen, daß eine Nullmove-Suche einen Wert größer dieser Schwelle erreichen kann.
Falls die Nullmove-Suche ausgeführt wird und einen Beta-Schnitt verursacht, ist es, neben dem Fall der nicht mehr benötigten Verifikation, immer dann zulässig abzubrechen, wenn das Ergebnis der Nullmove-Suche ein Schachmatt für den Gegner darstellt. Dieses Schachmatt einer Nullmove-Suche würde immer durch eine normale Suche bestätigt. Ist es nach einem Beta-Schnitt nicht möglich abzubrechen, so wird die Tiefe der Suche, abhängig von ihrem aktuellen Wert, um 1 oder 2 Halbzüge reduziert, und die Suche wird als Fail High des Verified Nullmove Pruning markiert.

Um **Zyklen** im Ablauf der Suche erkennen zu können, müssen Knoten markiert werden, die im aktuellen Suchpfad geöffnet wurden. Dies geschieht vor dem ersten Aufruf eines Kindknotens.

Im Ablauf der `NegaScout` - Funktion wird als nächstes geprüft, ob **Internal Iterative Deepening** angewendet werden soll. Dies ist dann der Fall, wenn kein bester Zug aus der Transposition Table bekannt ist und eine Suchtiefe größer zwei verbleibt. Falls dies zutrifft, wird die aktuelle Suchtiefe um zwei vermindert und die Suche wird ausgeführt. Nach ihrer Beendigung wird der gefundene beste Zug als Transposition Table - Zug markiert und die Suche wird zur vollen Tiefe wiederholt.

Zu Beginn der eigentlichen **Suche** der Kindknoten wird als erstes, falls vorhanden, immer der Zug aus der Transposition Table untersucht. Dies hat außer der Annahme einer guten Vorsortierung einen zweiten positiven Effekt: Falls diese Suche ein genügend hohes Ergebnis für einen Beta-Schnitt liefert, so kann man an dieser Stelle abbrechen, ohne eine Zugerzeugung für diese Position durchgeführt zu haben (außer im Falle des IID). Die Zugerzeugung ist eine der aufwendigsten Funktionen im Suchablauf, deshalb lohnt es sich, ihren Aufruf zu vermeiden.

```
//Search TT Move
if (tt_move != NOMOVE) {

   Move(n, tt_move, player, FALSE);    //Ausführung des Zuges ohne
                                       //Legalitätsprüfung(s. 2.1.6.)

   //NegaScout Search
   merit = -NegaScout(n, depth-1, -beta, -alpha, verify, -player);

   //Beta Cutoff?
   if (merit >= beta) {

      Store(n.zobrist, merit, best_tt_move, depth, LBOUND);//TT speichern

      if (!nullmove) SetKiller(&d, &best_move, &merit);    //Killer Heuristic
                                                           //speichern
      return merit;
   }
}
```

Konnte kein Beta-Schnitt erreicht werden, so müssen nun die möglichen Züge erzeugt werden:

```
//Generate Pseudo-Legal Movelist
int moves[];
char num = GetMoves(n, moves, player);
```

Die **Zugliste** wird dabei bereits während ihrer Erzeugung vorsortiert. Dies erfolgt zum einen nach dem MVV/LVA Kriterium. Die Implementation in Nemesis weicht dabei leicht von dem dargestellten Ansatz ab: Die verwendete Annahme ist, daß die Differenz zwischen der schlagenden und der geschlagenen Figur das zu optimierende Kriterium darstellt (s. 4.5.2.). Desweiteren werden die Züge nach den Daten der Killer Heuristic sortiert, falls ein Zug als Killer-Zug bekannt ist, bekommt er eine Position ganz vorne in der Zugliste.

Nun erfolgt die **Suche** der verbleibenden Zugmöglichkeiten, bei der zuerst geprüft werden muß, ob bereits ein Zug aus der Transposition Table untersucht wurde. Wenn ja, kann man versuchen, Alpha zu erhöhen, außerdem wird das Suchfenster für die folgenden Knoten auf eine Nullfenstersuche angepaßt. Danach wird die Zugliste iteriert: Zuerst wird festgestellt, ob es sich bei dem aktuellen Zug nicht um den Zug aus der Tranposition Table handelt, da dieser ja bereits untersucht wurde. Als nächstes wird der Zug ausgeführt und dabei seine Legalität geprüft, um, wie in 2.1.6. beschrieben, nur tatsächlich legale Positionen zu erzeugen. Dies ist unbedingt notwendig, um die in Kapitel 4.1. dargestellte Anforderung nach einer regelkonformen Spielweise zu erfüllen. Waren diese Prüfungen erfolgreich, so kommt Futility Pruning zum Einsatz. Sollten die notwendigen Vorbedingungen erfüllt sein (s. 2.2.10.), so kann der Kindknoten von der Suche ausgeschlossen werden, ansonsten wird er mit einem rekursiven Aufruf der Suchfunktion durchsucht.
Dieser Ablauf wird für alle möglichen Züge wiederholt, bis die Zugliste komplett iteriert wurde, ein Beta-Schnitt erzeugt werden konnte oder ein Schachmatt gefunden wurde:

```
//Adjust Bounds
short int a = MAX(alpha, merit);
short int b = (best_move == NOMOVE) ? beta : a + 1;

//Recursive Search
unsigned char i = 0;
short int m;
                                                    //Abbruchbedingungen
while (i < num && merit < beta && merit < -MATE_THRESHOLD) {

   Node t = n;
                                                    //Ausführung des Zuges mit
                                                    //Legalitätsprüfung
   if (moves[i] != tt_move && Move(t, moves[i], player, TRUE)) {

      //Futility Pruning
      if (depth != 1 || t.check || best_move == NOMOVE || ((player *
      (t.material[0] + t.material[1])) + MAXEVAL) > a) {

         //NegaScout Search
         m = -NegaScout(t, depth-1, -b, -a, verify, -player);

         //NegaScout Research?
         if (b != beta && m < beta && m > a && depth > 2) {
            m = -NegaScout(t, depth-1, -beta, -m, verify, -player);
         }

         //Best Move?
         if (m > merit) {
            merit = m;
            best_move = moves[i];
         }
```

```
            //Adjust Bounds
            a = MAX(a, merit);
            b = a + 1;
         }
      }

      i++;                        //Nächster Zug
   }
```

Ist die Untersuchung der Kindknoten beendet, so wird geprüft, ob es sich vorher um einen Fail High der Nullmove-Suche gehandelt hat. Wenn ja, so muß überprüft werden, ob das Ergebnis der Suche denn auch tatsächlich größer Beta ist, die Nullmove-Suche also bestätigt werden konnte. Ist dies nicht der Fall, so handelt es sich um eine Zugzwang-Situation (s. 2.2.8.) und die Suche muß mit der vollständigen Tiefe, also ohne die durch Verified Nullmove Pruning angewendete Tiefenreduktion, wiederholt werden.

Ist die Suche endgültig beendet, muß die Erkennung von Zyklen im Suchgraphen abgeschlossen werden, da der aktuelle Knoten jetzt verlassen wird.

Jetzt kann das Suchergebnis verarbeitet werden: Es wird geprüft, ob der aktive Spieler keinen legalen Zug hatte. Wenn ja, wird geprüft, ob er im Schach steht, ist dies der Fall, handelt es sich um ein Schachmatt, andernfalls um ein Patt. Für diese beiden Fällen wird das Ergebnis angepaßt, für ein Patt wird der Wert 0 zurückgegeben, für ein Schachmatt verwendet Nemesis den Wert -30000, der das schlechtestmögliche Ergebnis für den aktiven Spieler darstellt. Bei einem Schachmatt wird zu diesem Wert die aktuelle Tiefe im Suchbaum addiert, um eine Niederlage für den aktiven Spieler zu verbessern, je weiter sie von der Wurzel entfernt ist. Außerdem wird geprüft, ob das Ergebnis für die Killer Heuristic verwendet werden kann (s.o.). Dann wird das Suchergebnis in der **Transposition Table** gespeichert:

```
   //Result Processing
   if (best_move == NOMOVE) {
      if (n.check) merit = MATE + d;      //Schachmatt
      else merit = DRAW;                  //Patt
      best_move = NOMOVE;                 //kein bester Zug (s. 2.1.2.)
      tt_flag = VALID;                    //exaktes Ergebnis

   } else if (merit <= alpha) {
      best_move = NOMOVE;                 //kein bester Zug
      tt_flag = UBOUND;                   //Obergrenze

   } else if (merit >= beta) {
      tt_flag = LBOUND;                   //Untergrenze
                                          //Killer Heuristic speichern
      if (!nullmove) SetKiller(&d, &best_move, &merit);

   } else {
      tt_flag = VALID;                    //exaktes Ergebnis
                                          //Killer Heuristic speichern
      if (!nullmovem) SetKiller(&d, &best_move, &merit);
   }

   //TT Store
   Store(&(n.key), &(n.lock), &merit, &best_move, &depth, &tt_flag);

   return merit;
```

Nun kann das Suchergebnis der `NegaScout` - Funktion zurückgegeben werden.

4.5. Implementation weiterer Verfahren

4.5.1. Eröffnungsbibliothek

Eröffnungsbibliotheken sind im Internet in großer Zahl unter der GNU-Lizenz verfügbar [Rebel]. Ursprünglich hatte ich eine Bibliothek ausgewählt, die über 130.000 Stellungen enthält, diese Stellungen entsprechen einer Vielzahl von bekannten Eröffnungsvarianten, die im Maximum 25 Halbzüge lang sind. Diese Bibliothek ist jedoch leider für eine Verwendung in Nemesis nicht geeignet, da der iPAQ H5500 über 50 Sekunden braucht, um sie komplett einzulesen. Ich habe also einen Parser für diese Bibliothek geschrieben, um die Menge der Stellungen und damit die zum Einlesen benötigte Zeit zu reduzieren. Ich habe versucht, die volle Breite des Buchs zu erhalten, also die Menge der bekannten Varianten nicht zu reduzieren. Um dies zu erreichen, habe ich eine Maximaltiefe eingeführt, bis zu der verschiedene Zugabfolgen in der Bibliothek enthalten sein dürfen. Auf diese Weise ist es gelungen die Bibliothek von über 9 Megabyte auf etwa 2,3 Megabyte zu reduzieren und die zum Einlesen nötige Zeit auf ca. 10 Sekunden zu verringern. Es sind immer noch alle Eröffnungsvarianten vorhanden, jedoch nur noch bis zu einer Maximaltiefe von 10 Halbzügen. Dies sollte ausreichend sein, um Nemesis in eine akzeptable Position für das Mittelspiel zu bringen.
Nemesis liest die Eröffnungsbibliothek zu Beginn des Spiels ein und speichert die gefundenen Stellungen in der Transposition Table. Diese Einträge werden mit einem eigenen Flag als Daten der Bibliothek markiert. Sollte zu Beginn einer Suche ein solcher Eintrag gefunden werden, so wird der gespeicherte beste Zug ausgespielt, ohne eine tatsächliche Suche durchzuführen. Sollte die aktuelle Stellung nicht mehr im Buch gefunden werden, so wird eine normale MinMax-Suche gestartet. Während dieser Suche werden Einträge der Bibliothek in der Transposition Table überschrieben, da es unwahrscheinlich ist, nach dem Verlassen einer bekannten Variante im späteren Verlauf des Spiels wieder eine Stellung zu erreichen, die in der Bibliothek enthalten ist.
Die Abbildungen zeigen die verwendete Eröffnungsbibliothek im Einsatz: Das Spiel wurde gerade begonnen, Nemesis spielt mit den weißen Figuren. Man erkennt, daß keine Suche erfolgt ist, da erst ein Knoten geöffnet wurde:

Abb. 4.6: Eröffnung des Spiels aus der Bibliothek

Die dargestellte Hauptvariante entspricht der aus der Bibliothek bekannten Eröffnung: Der Zug 1. d2d4 wurde ausgespielt, die Bibliothek enthält als besten Folgezug die Variante 1. ... g8f6. Sollte

dieser Zug nun vom anderen Spieler gewählt werden, so wird diese Variante weiter verfolgt:

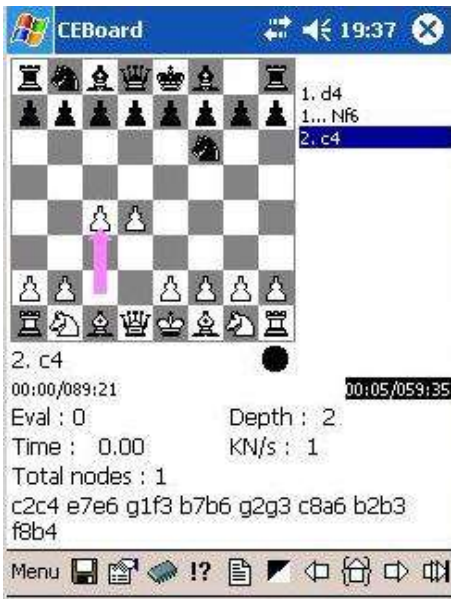


Abb. 4.7: Verlauf der Eröffnung

Der Gegenspieler hat 1. ... g8f6 gespielt, der weitere Verlauf dieser Variante (s. Abb. 4.7) hat sich nicht geändert. Sollte der andere Spieler jedoch eine andere Variante wählen, so hängt der weitere Ablauf davon ab, ob diese Variante ebenfalls in der Eröffnungsbibliothek gespeichert ist. Im dargestellten Spiel hat Weiß 2. c2c4 gespielt, eine mögliche Antwort hierauf wäre der vorgeschlagene Zug 2. ... e7e6. Schwarz spielt jedoch einen anderen Zug, nämlich 2. ... b8c6. Dieser Zug ist ebenfalls in der Bibliothek enthalten, was zu folgender Situation führt:

Abb. 4.8: Änderung der Eröffnungsvariante

Weiß spielt nun mit 3. g1f3 weiter, da dies der beste Folgezug auf 2. ... b8c6 ist, die Eröffnungsvariante hat sich also geändert. Man erkennt an diesem Beispiel die Funktionsweise der Eröffnungsbibliothek: Solange als gut bekannte Eröffnungen gespielt werden, sollten diese in der Bibliothek enthalten sein, um, wie oben zu sehen, Kenntnisse über den weiteren Verlauf zu haben.

4.5.2. Zugerzeugung und Vorsortierung

Die vielen verschiedenen Möglichkeiten der Zugerzeugung für ein Schachprogramm richten sich primär nach der verwendeten Datenstruktur zur Repräsentation des Spielfelds [Eppstein]. Da ich eine möglichst einfache Repräsentation für interne Datenstrukturen gewählt habe (4.4.1.), habe ich auch die Repräsentation des Spielfelds so einfach wie möglich gehalten, es wird als Array mit 64 Feldern dargestellt. Da keine Bitboards verwendet werden, müssen mögliche Züge als Indexwerte dieses Arrays bestimmt werden.

Die Zugerzeugung für Bauern wird **inkrementell** gehandhabt: Die aktuelle Position eines Bauern auf dem Spielfeld wird geprüft und aus dieser Information wird errechnet, welche Züge für den Bauern möglich sind. Ein Zug um ein Feld nach vorne ist immer dann möglich, wenn dieses Feld nicht besetzt ist. Ein Zug um zwei Felder nach vorne ist möglich, wenn die oben genannte Bedingung erfüllt ist, das Feld zwei Felder voraus ebenfalls nicht besetzt ist und der Bauer sich auf seiner Startposition befindet (s. Anhang D). Ein Schlagzug für Bauern ist immer dann vorhanden, wenn der Bauer einen Zug um ein Feld nach vorne und ein Feld nach links oder rechts durchführen kann, ohne die Grenzen des Spielfelds zu verletzen; außerdem muß dieses Feld natürlich von einer gegnerischen Figur besetzt sein.

Die Zugerzeugung der anderen Figuren baut auf **vorberechneten Arrays** auf. Die Züge, die in diesen Arrays enthalten sind, teilen sich in vier verschiedene Gruppen auf: Springerzüge, Läuferzüge, Turmzüge und Königszüge. Die Züge aller Figuren können aus diesen Arrays erzeugt werden, indem die aktuelle Figur und ihre Art zu ziehen berücksichtigt wird. Für Springer ist dies offensichtlich, da sie sich anderes bewegen als alle anderen Figuren. Für sie sind alle möglichen Züge, betreffend jeder möglichen Startposition auf dem Spielfeld, in einem Array enthalten, für Läufer und Türme gilt das gleiche. Die Zugerzeugung für die Dame basiert auf den Arrays der Läufer- und Turmzüge, da diese Zugarten zusammen den Zugmöglichkeiten der Dame entsprechen. Analog könnte man für den König verfahren, dessen Züge gegenüber einer Untermenge der Läufer- und Turmzüge redundant sind, ich habe sie jedoch aus implementationstechnischen Gründen gesondert behandelt.

Ich habe die Zugerzeugung und die **Vorsortierung** durch MVV/LVA sowie Killer Heuristic in einer Funktion integriert. Dieser Ansatz erlaubt es, die spätere Position eines Zuges in der Zugliste bereits im Moment seiner Erzeugung zu bestimmen. Dieser Ansatz ist effektiv, da nur ein einziger Durchlauf über alle Figuren notwendig ist, um sowohl die möglichen Züge als auch ihre Positionen zu bestimmen. Start- und Zielfeld eines Zuges können in je 6 Bit dargestellt werden, da ich einen 16 Bit – Wert pro Zug verwende, verbleiben 4 Bit, also der Wertebereich 0 bis 15, um die Position eines Zuges festlegen zu können. 0 entspricht hierbei einem Zug, der nicht vorsortiert werden soll, der Wert 15 entspricht einer Position ganz vorne in der Zugliste.
Jeder Zug wird bei seiner Erzeugung auf das Kriterium der Killer Heuristic geprüft, sollte er in dieser enthalten sein, so erhält er die Position 15. Sollte er nicht in der Killer Heuristic enthalten sein, hängt seine Position davon ab, ob es sich um einen Schlagzug handelt. Ist dies nicht der Fall, so bekommt er die Position 0, ist also in der Zugerzeugung nicht zu sortieren. Handelt es sich um einen Schlagzug, so bekommt der Zug die Position 8, also genau in der Mitte des Wertebereichs der zu sortierenden Züge. Diese Position wird nun mit dem Ergebnis des MVV/LVA – Kriteriums verrechnet: Hierzu bestimme ich die Differenz der Materialwerte der zu schlagenden und der schlagenden Figur und bilde diesen Wert auf den Bereich -7 bis +7 ab, wobei ausgeglichene Materialwerte dem Wert 0 entsprechen. Der Wert dieser Berechnung wird nun zu der vorherigen Position addiert, was dazu führt, daß Schlagzüge den Wertebereich 1 bis 15 einnehmen können, sortiert nach der Differenz der beiden beteiligten Figuren.

Die Sortierung der Züge erfolgt nach der Zugerzeugung. Die Zugliste wird nach Art des Selection Sort sortiert, allerdings mit einem wichtigen Unterschied zum normalen Verfahren: Sollte die größte gefundene Position in der Menge der unsortierten Züge den Wert 0 haben, so kann die Sortierung an dieser Stelle abgebrochen werden. Da, wie oben dargestellt, nur ein Teil der Züge eine Position größer 0 erhält, handelt es sich also tatsächlich um eine **partielle Sortierung**, die mit deutlich weniger Aufwand, als für einen vollständigen Selection Sort notwendig ist, bewältigt werden kann.

4.5.3. Transposition Table

Die Transposition Table wird intern durch zwei Arrays realisiert, diese Arrays haben jeweils eine Größe von 524288 (= 2^{19}) Einträgen. Eine Größe 2^n wurde gewählt, da sie einen einfachen Array-Zugriff über einen Index-Wert ermöglicht, der durch Anwendung eines Bitshift aus dem Hashwert des Knotens errechnet werden kann: Der Hashwert wird durch zwei 32-Bit Schlüsseln repräsentiert, diese Schlüssel werden addiert, dann wird das Ergebnis um 13 Bit (= 32 Bit – 19 Bit) nach rechts geschoben. Nun hat man einen Index-Wert im Bereich von 0 bis 2^{19}, mit dem direkt auf den Array zugegriffen werden kann. Da in den den Einträgen der Transposition Table die XOR-Verknüpfung der beiden Hashwerte gespeichert wird, kann man über diesen Wert prüfen, ob es sich um eine Kollision handelt; dies ist notwendig, da der Index-Wert durch die Anwendung des Bitshifts hierfür nicht mehr genügend Genauigkeit bietet. Eine Größe der Arrays größer 2^{19} war nicht verwendbar, da hierfür die Speicherressourcen des HP iPAQ H5500 nicht ausreichen. 524288 Einträgen sind zwar ausreichend, um einen Effizienzgewinn durch die Verwendung einer Transposition Table zu erreichen, mehr mögliche Einträge würden die Suche jedoch weiter beschleunigen. Die Anforderung 1.4 nach einer großen Speicherumgebung kann also nur bedingt als erfüllt betrachtet werden.
Die Implementation verwendet aus folgenden Gründen ein **zweistufiges Verfahren**: Im ersten Array werden Ergebnisse abhängig von ihrer Suchtiefe gespeichert, d.h. vorhandene Ergebnisse werden nur ersetzt, wenn die ersetzende Stellung gleich tief oder tiefer untersucht wurde. Dies dient dazu, einen maximalen Informationsgehalt (bezogen auf die Suchtiefe der Ergebnisse) dieses Arrays zu gewährleisten. Um nun zu verhindern, daß im Laufe mehrerer Suchen in diesem Array zu viele alte und damit wahrscheinlich nicht mehr benötigte Ergebnisse verbleiben, habe ich einen Alterungsfaktor eingeführt, der dazu dient, Ergebnisse nach Ablauf einer gewissen Zeit auch dann als ersetzbar zu bewerten, wenn die dazu eigentlich notwendige Suchtiefe nicht erreicht wurde.
Der zweite Array verfährt nach einem Immer-Ersetzen - Schema, d.h. jeder neue Eintrag überschreibt einen möglicherweise an der Position bereits vorhandenen Eintrag. Diese Ersetzungsstrategie stellt aktuelle lokale Informationen zur Verfügung, zu denen auch die meisten Zugumstellungen gehören. Dieser kombinierte Ansatz ist für eine Tranposition Table sehr gut geeignet, da sich beide Ersetzungsstrategien optimal ergänzen: Es werden sowohl Einträge mit möglichst hohem Informationsgehalt als auch Einträge mit hoher Lokalität, d.h. hoher Wahrscheinlichkeit einer Zugumstellung (2.1.8.), berücksichtigt.
Verschiedene Möglichkeiten der Implementation einer Transposition Table hat Sashi Lazar in [Lazar 95] ausführlich analysiert und ist dabei zu ähnlichen Schlüssen gekommen: Eine zweistufige Implementation ist, gerade bei eher geringer Größe der Transposition Table, einem einstufigen Ansatz überlegen. Die Ersetzungsstrategie spielt in diesem Zusammenhang ebenfalls eine große Rolle, und sollte primär Einträge mit hoher Suchtiefe, also hohem Informationsgehalt, berücksichtigen. Insgesamt kommt Lazar zu dem Schluß, daß eine Transposition Table die Menge der zu untersuchenden Knoten um bis zu 95% reduzieren kann. Eine möglichst effiziente Implementation dieses Verfahrens ist also gerade in ressourcenarmen Umgebungen wichtig, um die Suche zu beschleunigen.

4.5.4. Search Extensions

Um einen selektiven Anteil in die Suche zu integrieren, werden Search Extensions verwendet. Jeder Zug wird bei seiner Ausführung auf verschiedene Kriterien überprüft, sollte ein Kriterium erfüllt sein, so wird der `extension` – Wert des zu erzeugenden Knotens erhöht. Der `extension` – Wert gibt die spätere Erhöhung der Suchtiefe in Vierteln eines Halbzuges an. Zu Beginn der Untersuchung eines Knotens wird ausgewertet, ob der `extension` – Wert größer einem oder mehrerer Halbzüge ist. Wenn ja, wird die lokale Suchtiefe um diesen Wert erhöht, ein möglicher Rest verfällt. Falls der `extension` – Wert kleiner einem Halbzug ist, so wird er im weiteren Verlauf der Suche an die Kindknoten weitergegeben; er verfällt also nicht, sondern kann dazu beitragen, in Unterbäumen Extensions auszulösen. Es werden Bruchteile eines Halbzuges verwendet, um das Auftreten durch verschiedene Kriterien ausgelöster Extensions besser gegeneinander gewichten zu können.
Search Extensions haben natürlich auch einen negativen Einfluß auf die Performanz, da bestimmte Suchpfade tiefer untersucht werden müssen als andere. Die Vertiefung der Suche über eine gewisse Maximaltiefe wird also verhindert (s. 4.4.3.), um zu großen Aufwand zu vermeiden; außerdem kann man so möglichen Inkonsistenzen (s. 2.1.9.) vorbeugen.

Ich werde nun die einzelnen Kriterien vorstellen, die als Search Extensions verwendet werden. Diese Kriterien versuchen die Suchtiefe immer dann zu erhöhen, wenn es wahrscheinlich ist, daß in der Folge des untersuchten Knotens wichtige Veränderungen in der Spielsituation entstehen:

Das wichtigste Kriterium ist die **Check Extension**. Sie wird immer dann ausgelöst, wenn der aktive Spieler aus einem Schachgebot entkommen muß. Es gibt natürlich Züge, die Schach stellen und dennoch keinerlei Gefährdung bedeuten, deshalb habe ich versucht dieses Kriterium weiter zu differenzieren: Sollte der aktive Spieler seinen König bewegen müssen, um aus dem Schach zu entkommen, so wird der `extension` – Wert um 4 erhöht, dies entspricht der Erhöhung der Suchtiefe um einem Halbzug. Dieser Fall ist für den aktiven Spieler der gefährlichste, da den König zu bewegen immer die letzte Wahl ist, um ein Schachgebot aufzuheben. Oft entstehen aus solchen Situationen gefährliche Angriffskombinationen, die den König zwingen, seine Deckung endgültig zu verlassen. Sollte der aktive Spieler eine Figur dazwischen ziehen können, um das Schachgebot aufzuheben, so entspricht dies einer Erhöhung des `extension` – Wert um 3. Für den Fall, daß die Figur, die Schach gibt, geschlagen werden kann, wird der `extension` – Wert um 2 erhöht.

Die **Recapture Extension** geht von folgender Annahme aus: Sollte sich innerhalb der Suche ein Schlagzug ereignen, der zu dem gleichem Materialwert führt wie dem an der Wurzel, so weiß man, daß im Suchpfad zum aktuellen Knoten gleichwertige Figuren gegeneinander getauscht wurden. Ein Figurentausch kommt relativ oft vor und ist meistens Teil einer längeren Kombination verschiedener Züge. Da solche Kombinationen nur sehr wenige gute Zugmöglichkeiten offen lassen, kann man davon ausgehen, daß der letzte Schlagzug auf diese Art erzwungen wurde. Schlagkombinationen haben also ein hohes Gefährdungspotential, weil sie Züge erzwingen können, deswegen wird im hier dargestellten Fall die Recapture – Extension ausgelöst, die den `extension` – Wert um 3 erhöht.

Ein weitere Extension, die auf mögliche Gefährdungen des aktiven Spielers reagieren soll, ist die **Mate Threat Extension**. Sie wird immer dann ausgelöst, wenn die Nullmove – Suche des Verified Nullmove Pruning ein Schachmatt gegen den suchenden Spieler zurückgibt. Dieses Schachmatt ist nicht mit einem Schachmatt der normalen Suche gleichzusetzen, da es nur zustande kommt, weil der aktive Spieler das Zugrecht abgegeben hat. Nichtsdestoweniger reflektiert es eine deutliche Gefährdung der eigenen Stellung, da der andere Spieler Mattsetzen könnte, wenn er denn zweimal hintereinander ziehen dürfte. Deshalb wird für den Fall einer Mate Threat Extension der Wert der

`extension` um 3 erhöht. Diese Extension unterscheidet sich insoweit von den anderen, da sie als einzige nicht bei der Ausführung eines Zuges erkannt werden kann. Sie wird erst nach Ausführung des Verified Nullmove Pruning ausgelöst, so daß die mögliche Erhöhung der Suchtiefe nicht für den Knoten selber, sondern für alle seine Kindknoten erfolgt.

Um mögliche gravierende Änderungen der aktuellen Spielsituation zu erkennen, ist es außerdem wichtig, auf Bauernumwandlungen zu reagieren (s. Anhang D). Hierzu werden zwei verschiedene Extensions verwendet: Die **Pawn Push Extension** wird ausgelöst, wenn ein Bauer ein Feld vor der möglichen Umwandlung steht. Ist das Feld der Umwandlung frei, so wird der `extension` – Wert um 4 erhöht, ist es besetzt, so erfolgt eine Erhöhung um 3. Sollte ein Bauer tatsächlich die Umwandlung erreichen, so wird die **Pawn Promotion Extension** angewendet, die den `extension` – Wert um 4 anhebt.

4.5.5. Die Bewertungsfunktion

Die Bewertungsfunktion gibt dem Schachprogramm eine Art Persönlichkeit: Die Auswahl und Gewichtung der zur Bewertung verwendeten Kriterien kann sowohl zu einer defensiven als auch zu einer offensiven Spielweise führen. Viele Schachprogramme geben dem Benutzer die Möglichkeit, aus einer Menge an simulierten Gegenspielern mit verschiedenen Charakteristika auszuwählen, die Unterschiede in der Spielweise entstehen aber tatsächlich nur durch eine Veränderung der Bewertungsfunktion.
Für die Entwicklung der Bewertungsfunktion für Nemesis 1.0 mussten verschiedene Faktoren berücksichtigt werden. Zum einen ist es wichtig darauf zu achten, daß die Bewertungsfunktion keinen zu großen Overhead erzeugt, da die Zielplattform nur wenig rechnerische Ressourcen anbietet. Es ist auf Faktoren zu verzichten, die nur mit großem Aufwand zu bewerten sind; dies bezieht sich vor allem auf alle Arten einer rekursiven Bewertung, wie sie beispielsweise in der SEE verwendet wird. Um den Aufwand der Bewertungsfunktion in einem akzeptablen Rahmen zu halten, habe ich nur Kriterien verwendet, die isoliert betrachtet werden können: Die einzelnen Faktoren können **sequentiell** berechnet werden, es ist also zur Bewertung einer Stellung nur eine Schleife über alle Figuren notwendig. Dieses Verfahren schließt Kriterien aus, die erst Informationen verschiedener Figuren benötigen, um dann einen Wert zu berechnen; diese Art der Bewertung wäre nicht in einem sequentiellen Ablauf über alle Figuren zu realisieren.
Gleichzeitig ist es notwendig, möglichst aussagekräftige Kriterien zu finden. Diese Notwendigkeit wird durch die Gegebenheiten der Zielplattform noch gesteigert: Da Nemesis in einer ressourcenarmen Umgebung keine hohen Suchtiefen erreichen kann, muß die Bewertungsfunktion ebenfalls eher „kurzsichtig“ sein. Es ist sinnlos, Kriterien zu implementieren, die eine langfristige Entwicklung der Stellung berücksichtigen, wenn diese Entwicklung in der Suche nicht nachvollzogen werden kann. Im Gegenteil sollte die Bewertungsfunktion in der von mir betrachteten Anwendung derart sein, daß sie bei geringen Suchtiefen bereits wichtige positionelle Entwicklungen erkennen und bewerten kann.

Wie in 2.1.10. bereits erläutert liegt der Bewertung einer Stellung der jeweilige Materialwert zugrunde. Der Wert einer Figur bestimmt sich dabei nach dem folgenden Schema:

Figur	*Materialwert*
Bauer	100
Springer	305
Läufer	315

Figur	*Materialwert*
Turm	500
Dame	900

Abb. 4.9: Materialwerte der einzelnen Figuren

Die Materialwerte[19] sind vor allem dazu geeignet, die Verhältnisse der Figuren untereinander zu bestimmen. Nach dem verwendeten Schema würde Nemesis z.B. jederzeit einen Turm gegen einen Läufer und einen Springer tauschen, da diese zusammen einen höheren Materialwert haben. Ich habe die Werte für Springer und Läufer etwas über den Wert von drei Bauern angehoben, da Leichtfiguren gerade bei begrenzten Suchtiefen deutlich einfacher wirkungsvoll einzusetzen sind als Bauern. Der Läufer ist etwas stärker als der Springer, dies soll einen Tausch Springer gegen Läufer motivieren, da Läufer vor allem in Endspielen stärker sind. Ansonsten entsprechen die Werte den aus der Schachliteratur bekannten Annahmen, z.B. in [Tarrasch 31].

Zu dem Materialwert wird das Ergebnis der positionellen Evaluation addiert. Um einen Übergang der drei **Spielzustände** Eröffnung, Mittelspiel und Endspiel zu ermöglichen, verwende ich zur Bewertung der positionellen Faktoren folgendes Verfahren: Es werden für jeden Faktor drei verschiedene Ergebnisse berechnet, die der Gewichtung dieses Faktors in den unterschiedlichen Spielzuständen entsprechen. Im Laufe der Evaluation werden also drei Evaluationsergebnisse erzeugt, die jeweils der Summe der Ergebnisse für einen Spielzustand entsprechen. Ist die Evaluation beendet, so werden diese drei Werte mit der aktuellen **Phase** verrechnet: Die Phase repräsentiert den Zustand, in dem sich das Spiel gerade befindet. Sie wird als Wert in einem Bereich zwischen 0 und 156 definiert. Der Wert 0 repräsentiert den Beginn der Eröffnung, 78 das Mittelspiel und 156 den Schluß des Endspiels. Die Phase bestimmt sich als Maximum zweier verschiedener Kriterien. Einerseits wird sie von der Anzahl der bisher gespielten Züge getrieben, die Eröffnung dauert maximal 19 Züge, in diesem Bereich wird die Phase pro gespieltem Halbzug um 2 erhöht, dann ist das Mittelspiel, also die Phase 78, erreicht. Das Mittelspiel währt höchstens bis zum 39. Zug, dann beginnt der Übergang ins Endspiel, wobei die Phase wiederum pro gespieltem Halbzug um 2 erhöht wird, nach 78 Zügen ist also auf jeden Fall das Endspiel erreicht. Andererseits ist die Phase abhängig vom vorhandenen Material, für je 100 Punkte Materialverlust (egal welchen Spielers), wird sie um 2 erhöht. Das bedeutet, daß spätestens bei Reduzierung des Gesamtmaterials auf die Hälfte des anfänglichen Wertes das Mittelspiel erreicht ist, eine volle Endspiel-Bewertung erfolgt nach diesem Kriterium, wenn nur noch ein Bauer auf dem Feld ist.
Die Verrechnung der verschiedenen Evaluationsergebnisse mit der Phase geschieht derart, daß für jede mögliche Phase die Gewichtungen einzelner Faktoren ineinander übergehen. Für eine Phase kleiner gleich 78, also dem Übergang von der Eröffnung ins Mittelspiel, werden nur die Evaluationsergebnisse dieser beiden Spielzustände berücksichtigt, für eine Phase größer 78 werden die Evaluationsergebnisse des Mittel- und Endspiels verwendet. Sollte ein Faktor für die beiden berücksichtigten Spielzustände die gleiche Gewichtung haben, so wird er auch für jede Phase ihres Übergangs mit dem gleichen Wert in das Ergebnis eingehen. Sollte ein Faktor für die beiden Spielzustände unterschiedlich gewichtet sein, so wird dies in der Verrechnung derart berücksichtigt, daß abhängig von der Phase ein **fließender Übergang** der beiden Werte entsteht. Wird ein Faktor z.B. nur für das Mittelspiel verwendet, so wird er bei der Phase 0 nicht berücksichtigt, da hier noch kein Mittelspielanteil vorhanden ist. Bis zur Phase 78 wird sich sein Wert sukzessive auf das mögliche Maximum erhöhen, von der Phase 78 bis zur Phase 156 wird sein Wert wiederum auf 0 absinken.

19 Der König hat keinen definierten Materialwert, da er nicht geschlagen werden kann (s. 2.1.6.)

Ich habe diesen Ansatz entwickelt, um die Eindeutigkeit möglicher Kriterien zu erhöhen: Es gibt Faktoren, die nur in bestimmten Spielzuständen sinnvoll sind, würde man diese Kriterien in einer Bewertungsfunktion für alle Spielzustände verwenden, so würden Ergebnisse entstehen, die wenig Aussagekraft besitzen. Nach meinem Verfahren kann man die Gewichtung einzelner Faktoren über den Ablauf des Spiels **regulieren**, was eine sehr viel höhere Genauigkeit der Ergebnisse zur Folge hat.

Ein wichtiger Punkt der ebenfalls bei der Entwicklung einer Bewertungsfunktion zu berücksichtigen ist, ist der Wertebereich möglicher Ergebnisse: Ist das maximale Ergebnis der positionellen Bewertung kleiner als der niedrigste vorhandene Materialwert (also der eines Bauern), so wird die Spielweise des Schachprogramms immer primär von der Materialentwicklung abhängen, und nur sekundär von den positionellen Faktoren. Ist der Wertebereich jedoch größer als dieser Materialwert, so können Materialentwicklungen von positionellen Faktoren **verdeckt** werden, was bedeutet, daß die Bewertung einer Stellung positiv sein kann, obwohl Material verloren wird. Dies ist bei Nemesis der Fall, der maximale Wert der positionellen Faktoren entspricht hier dem Wert von drei Bauern. Hierbei handelt es sich um einen theoretischen Wert, der nur dann erreicht wird, wenn ein Spiel aus positionellen Gründen schon so gut wie gewonnen ist. Im Normalfall befinden sich die Ergebnisse der positionellen Bewertung in einem Bereich zwischen 0 und 200.
Die Abwägung von Material und positionellen Faktoren ist äußerst schwierig, da Materialentwicklung ein fest definiertes Kriterium darstellt, wohingegen positionelle Entwicklungen nur durch Heuristiken bestimmt werden können. Um eine hohe Spielstärke zu erreichen, scheint es dennoch notwendig, positionelle Faktoren so hoch zu bewerten, daß sie negative Materialentwicklungen ausgleichen können. Wenn man z.B. das Kriterium des Freibauern berücksichtigt (s.u.), so wird schnell klar, daß es in der Tat Faktoren gibt, die stärker gewichtet werden müssen, als der Gewinn oder Verlust eines Bauern. Nur so ist sichergestellt, daß Nemesis positionelle Entwicklungen wirklich berücksichtigt, und nicht nur der statischen Spielweise folgt, die entsteht, wenn die Zugwahl ausschließlich von möglichem Materialverlust oder -gewinn bestimmt wird. Dies hatte ich im Anforderungskatalog als eigenes Kriterium für Nemesis 1.0 festgelegt.

Neben der Bewertung der positionellen Faktoren werden zwei weitere Kriterien einer Stellung berücksichtigt: Falls ein Spieler nicht mehr genügend Figuren hat, um den anderen Spieler Schachmatt zu setzen[20], so wird dies mit dem Wert eines halben Läufers bestraft, da die Partie für diesen Spieler nicht mehr gewonnen werden kann. Außerdem wird für mögliche sowie ausgeführte Rochaden ein Bonus vergeben: Sollte ein Spieler das Recht besitzen, beide Rochaden auszuführen, beträgt der Bonus 20 Punkte. Wurde eine Rochade ausgeführt, beträgt er 40 Punkte. Hat der Spieler das Recht auf eine Rochade verloren, so wird der Bonus um 10 vermindert, er beträgt also 0, falls keine Rochade mehr ausgeführt werden kann.

Ich werde in den folgenden Abschnitten die Faktoren der **positionellen Evaluation** anhand der einzelnen Figuren darstellen.

20 Nach den in 4.4.3. dargestellten Kriterien für ein Patt durch Materialmangel

1. Bauern

Bauern werden primär über eine **Piece Square Table** genannte Technik bewertet. Dies bedeutet, daß man zur Bewertung einer Figur eine Datenstruktur verwendet, die angibt, welchen Wert eine Figur dieser Art auf jedem möglichen Feld des Spielfelds hat:

```
//Pawn Piece-Square Table
const char PawnPosVal[64] =
{ 0,  0,  0,  0,  0,  0,  0,  0 ,
 50, 60, 60, 60, 60, 60, 60, 50 ,
 30, 40, 40, 40, 40, 40, 40, 30 ,
 18, 20, 22, 28, 28, 22, 20, 18 ,
  6, 10, 14, 20, 20, 14, 10,  6 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 };
```

Die Darstellung muß von unten nach oben gelesen werden, die zweite Reihe von unten entspricht der Startposition der weißen Bauern[21]. Man erkennt, daß weiße Bauern auf ihren Ausgangsfeldern die Bewertung null haben. Sollten sie einen Zug um ein Feld nach vorne machen, ist die Bewertung immer noch null. Erst wenn sie den vom Ausgangsfeld möglichen Zug über zwei Felder machen, kann eine positive Bewertung erreicht werden. Dies soll Bauern animieren, aus der Startposition möglichst diesen Zug zu machen, da er nur hier möglich ist und die Bauern aktiver ins Spielgeschehen eingreifen läßt. Außerdem unterscheiden sich die Bewertungen deutlich für die mittleren und die äußeren Bauern: Es ist generell wichtiger, das Zentrum des Schachbretts zu besetzen als die Peripherie, deshalb werden die Bauern am besten bewertet, die dieses Zentrum kontrollieren. Gleichzeitig werden die Bewertungen immer höher, je weiter ein Bauer in Richtung der gegnerischen Grundreihe vorgedrungen ist. Dies soll die Möglichkeit wiedergeben, daß ein Bauer auf der gegnerischen Grundreihe gegen eine andere Figur umgetauscht werden kann[22], was vom Materialwert her einem maximalen Gewinn von 800 Punkten entspricht[23]! Die Bauern auf den äußersten Linien werden hier etwas schlechter bewertet, da sie, vor allem in Endspielstellungen, schwieriger zu verteidigen sind. Sie haben also eine geringere Chance, tatsächlich die Umwandlung zu erreichen. Die Piece-Square-Tables für Bauern werden in allen Phasen des Spiels gleich gewichtet.

Ein sehr wichtiges Kriterium ist das des **Freibauern**: Ein Bauer ist dann ein Freibauer, wenn ihm kein gegnerischer Bauer auf der eigenen oder den angrenzenden Linien gegenübersteht. Dieser Bauer kann mit wenigen Zügen die gegnerische Grundreihe erreichen, da er nicht mehr von gegnerischen Bauern aufgehalten werden kann. Freibauern müssen hoch bewertet werden, da, wie oben dargestellt, der mögliche Gewinn durch eine Umwandlung sehr hoch ist. Freibauern werden umso besser bewertet, je näher sie dem Feld ihrer Umwandlung kommen. Dieser Wert ist im Mittelspiel 0, falls der Bauer noch auf seinem Ausgangsfeld steht, er erhöht sich bis zu einem Maximum von 100 Punkten für einen Bauern, der sich ein Feld vor der Umwandlung befindet. Im Endspiel sind Freibauern noch gefährlicher als im Mittelspiel, da es hier weniger gegnerische Figuren gibt, die sie aufhalten könnten. Deshalb erhöhen sich die Werte hier auf ein Minimum von 23 Punkten und ein Maximum von 140 Punkten.

Als nächster Faktor wird die **Deckung des Königs** durch eigene Bauern bewertet: Der König steht sicher, wenn er einen Schutzwall aus Bauern um sich hat. Bauern werden also immer dann belohnt,

21 Ich verwende zur Veranschaulichung ausschließlich die Piece-Square-Tables weißer Figuren, diese sind invers natürlich auch für schwarze Figuren vorhanden

22 Deshalb gibt es keine Bewertung für Bauern auf dieser Reihe

23 Der Bauer wird gegen eine Dame getauscht

wenn sie Felder direkt vor oder neben dem eigenen König einnehmen. Dieses Kriterium ist vor allem im Mittelspiel wichtig, da verhindert werden muß, daß offene Angriffslinien auf den eigenen König entstehen. Es wird für diesen Spielzustand mit Werten von 15 bis 25 Punkten pro Bauer berücksichtigt. Im Endspiel wird dieses Kriterium nicht mehr verwendet, da hier nur noch wenige Figuren auf dem Feld sind, die Gefährdung des Königs sinkt dadurch deutlich. Es ist sogar wichtig, dieses Kriterium hier nicht mehr zu verwenden, da der König seine Deckung im Endspiel verlassen muß, um aktiv am Spielgeschehen teilzunehmen.

Alle bisher dargestellten Kriterien werden nun durch eine Bewertung der **Bauernstruktur** modifiziert. Ein durch einen anderen Bauern gedeckter Bauer ist stärker, erstens weil er, da gedeckt, nicht einfach geschlagen werden kann, und zweitens, weil im Falle einer Schlagabfolge der zweite Bauer den Platz des ersten einnimmt, also dessen Bewertung auf ihn übergeht. Sollte ein Bauer also durch einen anderen gedeckt sein, so wird die Summe aller bisherigen Kriterien mit einem Faktor multipliziert, dieser Faktor beträgt 1.2 im Mittelspiel und 1.25 im Endspiel. In der Eröffnungsphase wird dieses Kriterium nicht verwendet, da es eine dynamische Entwicklung der Bauernstruktur hemmt.

Als letztes Kriterium wird festgestellt, inwieweit ein Bauer **isoliert** ist. Isoliert bedeutet, daß auf einer oder beiden benachbarten Linien kein eigener Bauer mehr vorhanden sind. Dieses Kriterium ist aus mehreren Gründen wichtig. Zum einen kann ein isolierter Bauer im gesamten Spielverlauf nicht mehr von einem eigenen Bauern geschützt werden[24], was eine große Schwächung des eigenen Spielaufbaus bedeutet. Zum Anderen verhindert die Bestrafung eines isolierten Bauern, daß Doppelbauern erzeugt werden: Als Doppelbauer bezeichnet man zwei Bauern auf der gleichen Linie; diese Bauern sind ebenfalls schwach, da sie sich nicht gegenseitig decken können und leicht anderen Figuren zum Opfer fallen, außerdem hindern sie sich am Fortschreiten in Richtung der gegnerischen Grundreihe. Wenn nun zwei Bauern gedoppelt werden, so sind sie ebenfalls auf mindestens einer Seite isoliert, da einer der Bauern eine angrenzende Reihe verlassen musste, um auf die Reihe des anderen Bauern zu gelangen. Die Isolation eines Bauern ist also immer problematisch und muß deshalb stark berücksichtigt werden: Die Abwesenheit eines benachbarten Bauern wird in allen Phasen des Spiels mit mindestens 30 Punkten bestraft, im Endspiel ist der Wert sogar noch höher, wird jedoch leicht verringert, wenn es sich bei dem isolierten Bauern um einen Freibauern (s.o.) handelt.

2. Springer

Springer werden nach drei verschiedenen positionellen Kriterien bewertet. Zum einen ist es wichtig einen Springer nicht am Rand des Spielfelds zu postieren, da sich dort die Menge seiner möglichen Züge drastisch reduziert. Ein **zentralisierter** Springer hat acht mögliche Züge, ein Springer in der Ecke des Spielfelds nur zwei. Um dieses Kriterium zu bewerten, werden alle möglichen Züge des Springers[25] von seiner aktuellen Position mit je 5 Punkten bewertet. Ein Springer in der Mitte des Feldes bekommt also eine Bewertung von 40 Punkten, ein Springer in der Ecke jedoch nur eine Bewertung von 10 Punkten. Dieses Kriterium ist in allen Phasen des Spiels gleich wichtig.

Ein zweites Kriterium für den Wert eines Springers ist seine Entfernung zum generischen König, im Englischen **King Tropism** genannt. Je näher ein Springer dem König kommt, desto gefährlicher ist er für die gegnerische Stellung, desto besser muß er also bewertet werden. Für den Springer errechne ich King Tropism als Manhattan-Distanz, d.h. als Summe der Abstände auf der X- und Y-Achse. Für jede Achse wird eine minimale Distanz von 2 angenommen, da es durch die spezielle

24 Außer es ereignet sich ein Schlagzug, der einen eigenen Bauern wieder auf eine benachbarte Linie bringt
25 Ohne Berücksichtigung der Legalität oder der Tatsache, ob das Zielfeld besetzt ist

Zugweise des Springers keinen weiteren Vorteil bringt, ihn weiter als 2 Felder an den gegnerischen König anzunähern. Die Manhattan-Distanz wird mit einem Faktor verrechnet und zur Bewertung vom aktuellen Wert der Evaluation abgezogen. In der Eröffnung beträgt dieser Faktor 2, im Mittelspiel 3 und im Endspiel 1. Diese Werte habe ich gewählt, da Angriffe auf den gegnerischen König zwar in allen Phasen des Spiels wichtig, jedoch nur im Mittelspiel wirklich entscheidend sein können.

Außerdem berücksichtige ich für alle Figuren ein Kriterium, das ich **King Pressure**, also „Druck auf den (gegnerischen) König“ genannt habe. Dieses Kriterium wird nur im Mittelspiel verwendet, es soll mögliche Angriffe auf die gegnerische Königsstellung positiv bewerten. Hierzu werden alle Züge einer Figur, in diesem Fall des Springers, ausgewertet. Jeder Zug, der die Figur in einen Umkreis von 2 Feldern um den gegnerischen König bringen würde, wird mit 7 Punkten bewertet. Sollte die Figur die Möglichkeit haben, auf ein Feld zu ziehen, das dem gegnerischen König direkt benachbart ist, so wird dies mit 10 Punkten belohnt. Die positive Bewertung dieser Züge repräsentiert das Potential einer Figur, eine Attacke auf die gegnerische Königsstellung auszuführen. Dieses Kriterium ist so gestaltet, daß es bei einer Attacke auf ein einzelnes Feld noch nicht ins Gewicht fällt, sollte aber z.B. ein Turm eine offene Linie in der Nähe des gegnerischen Königs kontrollieren, so akkumulieren sich die einzelnen Werte des King Pressure – Kriteriums und haben dadurch signifikanten Einfluß auf die Gesamtbewertung.

3. Läufer

Läufer werden wie Bauern über einen **Piece-Square-Table** bewertet:

```
//Bishop Piece-Square Table
const char BishopPosVal[64] =
{  8,  8,  8,  8,  8,  8,  8 , 8 ,
   8, 12, 16, 16, 16, 16, 12,  8 ,
   8, 14, 20, 20, 20, 20, 14,  8 ,
   6, 14, 18, 18, 18, 18, 14,  6 ,
   4,  8, 10, 10, 10, 10,  8,  4 ,
   2,  6,  6,  4,  4,  6,  6,  2 ,
   2,  6,  2,  2,  2,  2,  6,  2 ,
   0,  0,  0,  0,  0,  0,  0,  0 };
```

Diese Werte sollen einen Läufer motivieren, sich im Zentrum des Spielfelds bzw. in Richtung der gegnerischen Stellung aufzuhalten. Diese Positionierung ist vor allem im strategischen Sinne relevant und wird deshalb in allen Phasen des Spiels gleich gewichtet.

Desweiteren ist für den Läufer der Faktor **Mobilität** wichtig: Mobilität bedeutet, möglichst viele verschiedene Züge zur Verfügung zu haben, um die strategischen Möglichkeiten der Figur zur erhöhen. Mobilität wird berechnet, indem jeder mögliche Zug[26] mit 4 Punkten belohnt wird. Bei der Berechnung der möglichen Züge wird zusätzlich von folgender Annahme ausgegangen: Sollte sich auf einer Zugdiagonalen des Läufers eine eigene Figur befinden, die ebenfalls auf dieser Diagonalen ziehen kann, so wird die Zugerzeugung nicht abgebrochen, es wird also so berechnet, als wenn der Läufer durch diese Figur hindurch ziehen könnte. Dies nennt man ein X-Ray -Verfahren[27]. Es ist deshalb sinnvoll anzuwenden, da sich die beiden Figuren auf der Diagonalen verstärken, sie also dafür belohnt werden sollten. Für einen Läufer bezieht sich dieses Verfahren ausschließlich auf das Vorhandensein einer Dame auf einer Diagonalen, da ein eigener Läufer dort nicht postiert sein kann. Das Kriterium der Mobilität wird in der Eröffnung und im Mittelspiel

26 Legalität wird aus rechnerischen Gründen nicht berücksichtigt
27 X-Ray: Englisch für Röntgenstrahlung

angewendet. Im Endspiel sind nur noch wenige Figuren auf dem Feld, so daß Mobilität bereits dadurch gewährleistet ist.

Der Läufer wird ebenfalls für das **King-Pressure** - Kriterium belohnt, wobei dieses, wie die Mobilität, als X-Ray -Kriterium angewendet wird.

4. Turm

Türme werden für King Tropism und für Center Tropism bestraft. **King Tropism** wird als Manhattan-Distanz berechnet. In der Eröffnung wird dieses Kriterium nicht verwendet, da hier die Entwicklung anderer Figuren wichtiger ist, im Mittelspiel wird der King Tropism - Wert mit dem Faktor 3 multipliziert, im Endspiel mit dem Faktor 1.

Center Tropism bewertet den Abstand einer Figur vom Zentrum des Spielfeldes, da sie hier ihre größte Stärke entfalten kann. Dieser Wert wird sowohl als Maximaldistanz (Maximum der Abstände auf der X- und Y-Achse) als auch als Minimaldistanz (Minimum der Abstände auf der X- und Y-Achse) berücksichtigt: Die Maximaldistanz erhöht sich immer dann, wenn der Turm sich tatsächlich dem Zentrum des Spielfelds annähert, dieser Wert wird mit dem Faktor 2 multipliziert. Die Minimaldistanz erhöht sich bereits dann, wenn der Turm ein Reihe oder Linie besetzt, die näher am Zentrum vorbei führt als die vorherige. Dieser Bewertung liegt die Annahme zugrunde, daß der Turm sich verbessert hat, da er nun mit weniger Zügen das Zentrum erreichen kann; er wird mit dem Faktor 4 verrechnet. Center Tropism findet nur im Mittelspiel Anwendung, da die Positionierung des Turms in der Mitte des Spielfelds nur dann eine Verbesserung bedeutet, wenn noch viele andere Figuren auf dem Feld sind. In der Eröffnung wird es nicht verwendet, da hier andere Figuren wichtiger sind.

Türme werden außerdem dafür belohnt, auf einer **offenen Linie** zu stehen, da der Turm so aktiver ins Spielgeschehen eingreifen kann. Eine Linie ist immer dann offen, wenn sie nicht durch eigene Figuren blockiert ist. Sollte sich also auf der Linie eines Turms in Richtung des Gegners eine gegnerische oder gar keine Figur finden, so wird er mit einem Wert von 20 belohnt. Dieser Faktor muß im Endspiel nicht berücksichtigt werden, da die Grundannahme strategischer Natur ist, in Endspielen mit wenigen Figuren also keine Bedeutung mehr hat.

Ein Bonus wird für jeden Turm vergeben, der sich auf der **Grundreihe der gegnerischen Bauern** befindet. Von dort kann er sowohl die gegnerischen Bauern als auch die gegnerische Königsstellung gefährden. Dieses Kriterium wird in Eröffnung und Mittelspiel mit 15 Punkten belohnt, im Endspiel findet es keine Anwendung.

Ein weiteres Kriterium ist, ob Türme miteinander **verbunden** sind. Dies ist dann der Fall, wenn ein Turm durch einen möglichen Zug einen anderen eigenen Turm erreichen kann. Türme können also horizontal oder vertikal verbunden sein. Die Gewichtung dieses Kriteriums erfolgt sehr unterschiedlich: In der Eröffnung werden beide Arten der Verbindung mit 20 Punkten bewertet. Eine vertikale Verbindung ist immer stark, da die beiden Türme sich in Angriffen auf die gegnerische Stellung unterstützen. Eine horizontale Verbindung wird ebenfalls hoch bewertet, da sie in der Eröffnung bedeutet, daß alle anderen Figuren die Grundreihe verlassen haben und eine Rochade erfolgt ist, was eine gute Entwicklung der eigenen Stellung bedeutet. Im Mittelspiel werden vertikal verbundene Türme mit 30 Punkten bewertet, da sie in dieser Phase des Spiels ihren größten strategischen Wert haben. Eine horizontale Verbindung wird nur noch mit 10 Punkten bewertet, da sie nicht so wichtig ist. Im Endspiel werden beide Werte auf 10 reduziert, da hier andere Faktoren höhere Wertigkeit haben.

Auch Türme verwenden das **King Pressure** - Kriterium. Es wird hier ebenfalls als X-Ray - Verfahren verwendet, wobei für Türme die theoretische Möglichkeit berücksichtigt wird, durch andere Türme und Damen hindurchzuziehen.

5. Dame

In der Eröffnung ist es nicht sinnvoll, mit der Dame zu ziehen, da zuerst andere Figuren entwickelt werden müssen. Deshalb wird die Dame in dieser Phase mit 20 Punkten bestraft, falls sie ihre Ausgangsposition verläßt.

Für die Dame wird ein **Piece-Square-Table** verwendet, um ihren positionellen Wert zu ermitteln:

```
//Queen Piece-Square Table
const char QueenPosVal[64] =
{  0,  3,  6,  6,  6,  6,  3,  0 ,
   3,  6,  9, 12, 12,  9,  6,  3 ,
   6,  9, 12, 15, 15, 12,  9,  6 ,
   9, 12, 15, 18, 18, 15, 12,  9 ,
   9, 12, 15, 18, 18, 15, 12,  9 ,
   6,  9, 12, 15, 15, 12,  9,  6 ,
   3,  6,  9, 12, 12,  9,  6,  3 ,
   0,  3,  6,  6,  6,  6,  3 , 0 };
```

Diese Piece-Square - Werte sind vor allem dazu gedacht, eine Zentralisierung der Dame zu belohnen, da sie im Zentrum des Spielfelds ihre größte Stärke entfalten kann, und von dort die meisten Zugmöglichkeiten besitzt.

Die Dame wird für **King Tropism** bestraft, um mögliche Gefahren für die gegnerische Königsstellung zu bewerten. Dieses Kriterium wird in der Eröffnung nicht angewendet (s.o.), im Mittelspiel wird es mit dem Faktor 4, im Endspiel mit dem Faktor 2 verrechnet.

Außerdem wird für die Dame ebenfalls das **King Pressure** - Kriterium angewendet. Es wird als X-Ray - Verfahren benutzt, wobei die Dame in der Evaluation die Möglichkeit hat, durch eigene Läufer, Türme oder Damen hindurchzuziehen, und zwar in der Art, daß für horizontale Züge Türme und Damen und für diagonale Züge Läufer und Damen berücksichtigt werden.

6. König

Die Sicherheit des Königs wird durch die eigene Bauerndeckung sowie die King-Pressure der gegnerischen Figuren bestimmt. Positionelle Faktoren werden über einen **Piece-Square-Table** bewertet:

```
//King Piece-Square Table
const char KingPosVal[64] =
{ 0,  0,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  0,  0,  0,  0,  0,  0,  0 ,
  0,  5,  5,  0,  0,  5,  5,  0 ,
  5, 10, 10, 10, 10, 10, 10,  5 };
```

Diese Werte dienen dazu, den König innerhalb der eigenen Verteidigung zu halten, um ihn vor potentiellen Attacken zu schützen. Außerdem sollte er hier von der Deckung durch eigene Bauern profitieren können, vor allem dann, wenn eine Rochade erfolgt ist. Die Felder in den Ecken sind etwas schlechter bewertet, da hier ein Erstickungsmatt wahrscheinlicher ist.

Für die **Endspiel-Bewertung** verwende ich einen anderen Piece-Square-Table, um den König aktiv ins Spiel zu führen:

```
//King Piece-Square Table Endgame
const char KingEndPosVal[64] =
{ -4,  0,  4,  8,  8,  4,  0, -4 ,
   0, 10, 14, 16, 16, 14, 10,  0 ,
   4, 14, 20, 24, 24, 20, 14,  4 ,
   8, 16, 24, 30, 30, 24, 16,  8 ,
   8, 16, 24, 30, 30, 24, 16,  8 ,
   4, 14, 20, 24, 24, 20, 14,  4 ,
   0, 10, 14, 16, 16, 14, 10,  0 ,
  -4,  0,  4,  8,  8,  4,  0, -4 };
```

Da im Endspiel nicht mehr viele Figuren auf dem Feld sind, ist die Gefährdung für den König deutlich geringer. Es ist wichtig, den König hier in die Mitte des Spielfelds zu bringen, da er in Endspielen eine wichtige Figur ist, um eigene Bauern und andere Figuren zu unterstützen. Außerdem ist es bei geringem Gesamtmaterial schwieriger, den König in der Mitte des Spielfelds mattzusetzen, als am Rand.

5. Evaluation von Anwendung und Zielplattform

5.1. Ziele von Testverfahren

Ein Schachprogramm zu evaluieren ist eine schwierige Aufgabe, da es kein festes Maß gibt, an dem man es bewerten könnte. Für eine Stellung in einem Schachspiel gibt es nicht immer eine eindeutige Lösung, und wenn doch, so ist es aus Gründen des rechnerischen Aufwand oft nicht möglich, diese zu finden. Das Optimum ist nur dann erkennbar, wenn innerhalb des Suchhorizonts ein Schachmatt erreicht werden kann. Dies kann jedoch nicht das einzige Kriterium sein, an dem sich die Qualität eines Schachprogramms bemessen läßt. Nemesis muß genauso in der Lage sein, gute Züge zu finden, wenn das Spielende noch nicht absehbar ist. Es ist also notwendig, **relative Bewertungsmöglichkeiten** für ein Schachprogramm zu verwenden, da zum Gewinn einer Schachpartie mehr nötig ist, als ein mögliches Schachmatt zu erkennen, was das einzige absolute Kriterium darstellen würde. Nemesis sollte in jeder möglichen Stellung in der Lage sein, denjenigen Zug zu finden, der es erlaubt, eine gute Entwicklung des Spiels zu erreichen. Wenn dies gewährleistet ist, wird es umso einfacher, den Gegner irgendwann mattzusetzen.

Die Fähigkeit, gute Züge zu finden, ist von verschiedenen Faktoren abhängig. Zum einen gilt es den technischen Aspekt zu bewerten: Dies bezieht sich auf die **Ausführungsgeschwindigkeit**, also die Anzahl der untersuchten Knoten pro Zeiteinheit sowie die Anzahl benötigter Knoten pro Suchtiefe. Bei höherer Ausführungsgeschwindigkeit können höhere Suchtiefen erreicht werden, das Schachprogramm hat also mehr Informationen über den weiteren Verlauf des Spiels, was notwendigerweise eine Verbesserung der Spielqualität darstellt. Die Anzahl der Knoten pro Sekunde (eng: NPS, Nodes per Second) ist hierbei am einfachsten zu bewerten. Ein Schachprogramm ist unter diesem Aspekt besser, wenn es einen höheren NPS-Wert hat als ein anderes. Dieser Wert kann jedoch nicht isoliert betrachtet werden: Es kann nämlich durchaus sein, daß ein Schachprogramm aus algorithmischen Gründen einen niedrigeren NPS-Wert hat als ein anderes, gleichzeitig aber weniger Knoten pro Suchtiefe benötigt, im Endeffekt also trotzdem höhere Suchtiefen erreicht. Da diese Werte also keine absolute Aussagekraft besitzen, werde ich in meiner Bewertung vor allem Tests der relativen Geschwindigkeitsverbesserung durchführen, die durch verschiedene Verfahren möglich sind. Außerdem werde ich testen, welche unterschiedlichen Resultate in der Ausführungsgeschwindigkeit Nemesis auf dem Hewlett Packard iPAQ H5500 PDA, dem Pocket PC Emulator und einer Version speziell für den PC erreicht. Die Vergleiche dieser Ergebnisse können weiteren Aufschluß über die tatsächliche Leistungsfähigkeit der Zielplattform geben.

Zum anderen muß die **Spielstärke** des Schachprogramms bewertet werden. Mit Spielstärke bezeichnet man die Qualität der Zugwahl, also vor allem die Fähigkeit, in ausgeglichenen Situationen einen angemessenen Zug zu finden und die eigene Stellung möglichst zu verbessern. Hierfür gibt es ebenfalls kein festes Maß: Die Spielstärke kann allenfalls durch Resultate geschätzt werden, die gegen Gegner erreicht werden, deren Spielstärke bekannt ist. Es ist offensichtlich, daß es nicht möglich ist, auf einem PDA ein Schachprogramm zu entwickeln, das auf allerhöchstem Niveau spielen kann, da die rechnerischen Möglichkeiten der Plattform hierfür einfach zu beschränkt sind. Der Sinn einer Evaluation der Spielstärke kann also nur sein, herauszufinden, ob Nemesis gegen Spieler der Stärke eines erfahrenen Vereinsspielers bestehen kann, dann sollte immer in der Lage sein, gegen durchschnittliche Anwender zu gewinnen. Die Ergebnisse dieser Evaluation geben darüber Aufschluß, ob es möglich ist, auf einem ressourcenarmen System eine Schachanwendung zu implementieren, die eine praxistaugliche Spielstärke erreicht.

5.2. Testmöglichkeiten und Durchführung

Um die Ausführungsgeschwindigkeit sowie die Korrektheit der Ergebnisse sowie zu testen, werde ich **Teststellungen** verwenden. Hierbei handelt es sich um Sammlungen verschiedenster Stellungen, die aber alle eine eindeutige Lösung, im Sinne eines bestmöglichen Zuges, haben. Diese Lösung ist öffentlich bekannt und vielfach evaluiert. Man kann diese Stellungen verwenden, um zu testen, ob das eigene Schachprogramm in der Lage ist, den richtigen Zug zu finden. Sollte die Lösung gefunden werden, kann man anhand der notwendigen Knotenexpansionen sowie anderer Daten die Effizienz der Suche bewerten und feststellen, welche Verfahren die Geschwindigkeit der Suche effektiv erhöhen.
Ich werde für alle Tests die Testsammlung Win At Chess (WAC) verwenden, die im Internet frei verfügbar ist [Loiodice]. Diese Sammlung besteht ursprünglich aus über 300 Stellungen, die ich zu Testzwecken auf 100 reduziert habe. Da sich die Schwierigkeit der Stellungen über die gesamte Sammlung teils deutlich unterscheidet, habe ich willkürlich 100 Stellungen in Blöcken aus der Gesamtmenge entnommen. Die getesteten Stellungen können auf der beiliegenden CD im HTML-Format eingesehen werden. Für jede Stellung hat Nemesis 30 Sekunden Zeit, um die richtige Lösung zu finden. Kann die Lösung innerhalb dieser Zeit nicht gefunden werden, so wird die Suche abgebrochen und mit der nächsten Stellung fortgefahren. Die in 4.4.2. dargestellten Verfahren zur Zeitkontrolle finden in diesen Tests keine Anwendung. Die Transposition Table sowie die Killer – Heuristic werden für jede Stellung neu initialisiert, da Informationen der vorherigen Stellung für die nächste nicht effizienzsteigernd sein können. Ansonsten ist der Ablauf einer Suche analog zu der, die auch in der Spielversion Anwendung findet.

Um eine Bewertung der Spielstärke zu erreichen, werde ich eine Reihe von **Testspielen** gegen andere Schachprogramme durchführen, um diese als Referenz zu nutzen. Hiefür werde ich TSCP 1.81 [Kerrigan] und GNUChess 4 [GNU] verwenden, die aus mehreren Gründen hierfür gut geeignet sind. Zum einen handelt es sich um Open-Source - Programme mit einer langen Entwicklungsgeschichte, es ist also als sicher anzunehmen, daß sie keine größeren Fehler enthalten und als stabile Gegenspieler angesehen werden können. Zum anderen bieten diese beiden Programme einen guten Maßstab zur Bewertung der Spielstärke: Es gibt im Internet diverse Ligen, in denen Schachprogramme in einer Vielzahl von Spielen gegeneinander antreten. Hierbei wird ein sogenannter ELO-Wert berechnet, der die relative Spielstärke eines Programms angibt. TSCP 1.81 erreicht hier Ergebnisse von über 1800 ELO-Punkten [Dubois], dies entspricht der Spielstärke eines „Amateur, Klasse A, sehr guter Vereinsspieler“. GNUChess ist als eines der stärksten nicht-kommerziellen Produkte bekannt, die Version 4 besitzt einen ELO-Wert von 2075 Punkten [Chess War]. Dieser Wert liegt in der nächsten Stufe des ELO-Wertungssystems, die der Spielstärke eines „Meisteranwärters“ bzw. „Schach-Experten“ entspricht [Wikipedia Elo].
TSCP 1.81 besitzt in etwa die Spielstärke, die Nemesis erreichen sollte, um gute Gegenspieler zu schlagen und auch sehr gute Spieler langfristig zu fordern. GNUChess 4 stellt einen hohen Maßstab für die Bewertung der Spielstärke dar. Sollte Nemesis auf einem PDA in der Lage sein, diesem Gegner auf einem PC Widerstand zu leisten, so kann man davon ausgehen, daß es auch sehr gute menschliche Gegenspieler schlagen kann.

5.3. Ergebnisse von Teststellungen

5.3.1. Vollständige Optimierung

Weil, wie in 4.3.4. erläutert wurde, die implementierten Verfahren in einer MinMax-Suche miteinander interagieren, habe ich Nemesis zuerst mit allen Optimierungen getestet, um später vergleichen zu können, welche Nachteile entstehen, wenn man auf die Verwendung einzelner Verfahren verzichtet. Ich werde hier einmalig die Resultate eines Testdurchlaufs vollständig darstellen, um einen Eindruck der Testdaten und ihrer Menge zu vermitteln:

Stellung	Knotenexpansionen	Suchtiefe	Zeit(s)	NPS	Suchergebnis	bester Zug	Lösung gefunden
WAC001	12678	4	1	12678	29993	g3g6	X
WAC002	748201	8	31	24135	66	c4c3	
WAC003	3091	3	0	3091	189	e3g3	X
WAC004	210	2	0	210	29995	h6h7	X
WAC005	560	2	0	560	29995	c6c4	X
WAC006	794	3	0	794	406	b6b7	X
WAC007	1635	3	0	1635	495	g4e3	X
WAC008	341	2	0	341	216	h6f7	X
WAC009	4749	3	0	4749	0	d6h2	X
WAC010	5607	3	1	5607	168	h4h7	X
WAC011	2553	3	0	2553	364	f3c6	X
WAC012	1945	3	0	1945	29995	g4f3	X
WAC013	3955	3	1	3955	185	f1f8	X
WAC014	17991	4	1	17991	316	h3h7	X
WAC015	1048	3	0	1048	600	b8b7	X
WAC016	887	2	0	887	4	e2c3	X
WAC017	6956	3	1	6956	43	c4e5	X
WAC018	128686	8	5	25737	163	a8h8	X
WAC019	45216	4	2	22608	83	c5c6	X
WAC020	3240	3	0	3240	715	d7b5	X
WAC021	371	2	0	371	105	f5h6	X
WAC022	524267	7	31	16911	69	g5f7	
WAC023	47612	4	3	15870	290	g2g4	X
WAC024	742	2	0	742	4	g7d4	X
WAC025	131	2	0	131	800	g4h4	X
WAC101	1640	3	0	1640	26	d4c3	X
WAC102	581	2	0	581	29995	c8f8	X
WAC103	16750	3	2	8375	0	g1g6	X
WAC104	15092	3	1	15092	355	e2h5	X
WAC105	8498	3	1	8498	301	h5h4	X
WAC106	26438	4	1	26438	505	e4f2	X
WAC107	233	2	0	233	-45	d4b5	X
WAC108	61975	5	3	20658	321	c5e5	X
WAC109	850	2	0	850	-25	c4c3	X
WAC110	4383	3	0	4383	580	a7e3	X
WAC111	166715	6	7	23816	477	g1f1	X
WAC112	971	3	0	971	-80	e1e6	X
WAC113	130	2	0	130	126	g7f6	X

Stellung	Knotenexpansionen	Suchtiefe	Zeit(s)	NPS	Suchergebnis	bester Zug	Lösung gefunden
WAC114	30549	4	2	15274	106	d3h7	X
WAC115	3250	3	0	3250	77	e8d6	X
WAC116	519540	6	29	17915	142	d8d2	X
WAC117	488	2	1	488	379	d6e4	X
WAC118	48303	4	2	24151	28	f4h4	X
WAC119	17329	5	1	17329	187	d8d3	X
WAC120	292444	6	17	17202	212	g5g6	X
WAC121	10643	4	1	10643	762	c6f3	X
WAC122	6668	3	0	6668	848	d1f1	X
WAC123	14367	4	1	14367	195	b7d5	X
WAC124	60563	5	3	20187	160	g4g3	X
WAC125	1273	2	0	1273	305	b6d4	X
WAC126	1511	3	0	1511	800	f6c6	X
WAC127	127430	5	7	18204	380	b2b7	X
WAC128	2747	2	1	2747	-35	f7g6	X
WAC129	29003	4	2	14501	101	b7f3	X
WAC130	25657	4	1	25657	-75	f6h6	X
WAC131	610916	7	31	19706	112	d4f6	
WAC132	26893	5	1	26893	29993	e5e1	X
WAC133	27600	4	2	13800	-11	g3h4	X
WAC134	1736	2	0	1736	-139	d8d1	X
WAC135	4991	3	0	4991	-90	d7d4	X
WAC136	5436	2	0	5436	605	c1c8	X
WAC137	14352	3	1	14352	-24	d1d7	X
WAC138	197735	6	11	17975	959	h4h5	X
WAC139	351680	6	21	16746	29992	e4f6	X
WAC140	1244	2	0	1244	249	c3c7	X
WAC141	518394	6	31	16722	-63	g2f1	
WAC142	7231	3	0	7231	227	e1e8	X
WAC143	12730	4	1	12730	29993	g6h6	X
WAC144	185830	5	12	15485	219	d4d3	X
WAC145	599019	6	31	19323	127	f1f3	
WAC146	158883	8	5	31776	315	f5c8	X
WAC147	33484	4	3	11161	123	f6g4	X
WAC148	14535	4	1	14535	73	g1g7	X
WAC149	23998	4	1	23998	160	e4g2	X
WAC150	440899	6	23	19169	389	d6a3	X
WAC226	267029	6	15	17801	254	e5f7	X
WAC227	183677	5	11	16697	199	d6d5	X
WAC228	500632	5	31	16149	14	d1g4	
WAC229	721779	9	31	23283	76	e4d5	
WAC230	690568	9	31	22276	78	b7h7	
WAC231	187495	6	10	18749	84	c1g5	X
WAC232	62302	5	3	20767	-16	a6b5	X
WAC233	35008	4	2	17504	476	d4b3	X
WAC234	5058	3	0	5058	88	h3b3	X
WAC235	634481	7	31	20467	224	e6e4	

Stellung	Knotenexpansionen	Suchtiefe	Zeit(s)	NPS	Suchergebnis	bester Zug	Lösung gefunden
WAC236	1660	3	0	1660	52	c3c1	X
WAC237	565932	5	31	18255	-73	f7e8	
WAC238	575386	8	31	18560	110	g2b7	X
WAC239	620	2	0	620	-70	f2f1	X
WAC240	163292	6	9	18143	27	c2c6	X
WAC241	579687	6	31	18699	191	f5f6	
WAC242	379335	6	21	18063	318	d1d7	X
WAC243	586079	7	31	18905	94	b4b5	
WAC244	86895	6	5	17379	29987	e3c5	X
WAC245	533127	6	31	17197	128	c3b5	
WAC246	6776	3	0	6776	29995	g4h5	X
WAC247	656658	7	31	21182	40	d6c5	
WAC248	51026	5	2	25513	0	d6c5	X
WAC249	500326	6	31	16139	65	d4d5	
WAC250	3610	3	0	3610	21	e3e8	X

Abb. 5.1: Vollständige Resultate von Teststellungen mit allen Optimierungen

Im Weiteren wird eine akkumulierte Darstellung der Testresultate verwendet, da hier entscheidende Werte deutlicher hervortreten. Alle Resultate finden sich in ausführlicher Form auf der beiliegenden CD, da die Datenmenge zu groß ist, um sie in der Arbeit darzustellen.

Dateiname	WAC_100.epd
Anzahl Stellungen	100
Lösungen Gefunden	85
Erfolgsquote	85%
Gesamtzeit(s)	334
Anzahl Knotenexpansionen	14275511
NPS	42741
Durchschnittliche Zeit(s) (gelöste Stellungen)	3.40
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	62417
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	598004
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.76
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	6.73

Abb. 5.2: Akkumulierte Resultate von Teststellungen mit allen Optimierungen

Man erkennt, daß Nemesis mit allen Optimierungsmaßnahmen in der Lage ist, 85% der Teststellungen erfolgreich zu bewältigen. Die Lösungen konnten in durchschnittlich 3,4 Sekunden gefunden werden. Der durchschnittliche NPS-Wert liegt bei 42741; da im Mittel 62417 Knotenexpansionen benötigt wurden, um richtige Lösungen zu finden, ist offensichtlich, daß, wenn eine Lösung gefunden werden konnte, diese schnell erkannt wurde. Dies liegt vor allem an der Bewertungsfunktion, die die entscheidenden positionellen Entwicklungen bereits frühzeitig erkennt und entsprechend bewertet. Interessant ist der Vergleich dieser Werte zu einem Test ohne Bewertungsfunktion, bei dem die Bewertung von Blättern ausschließlich auf dem Materialwert beruht. Die Resultate dieses Tests zeigen außerdem eine gewisse Problematik, die mit einer Evaluation durch Teststellungen verbunden ist:

Lösungen Gefunden	81
Erfolgsquote	81%
Gesamtzeit(s)	379
Anzahl Knotenexpansionen	28218893
NPS	74456
Durchschnittliche Zeit(s) (gelöste Stellungen)	2.64
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	92854
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	1089351
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.91
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	7.11

Abb. 5.3: Resultate von Teststellungen ohne Bewertungsfunktion

Daß ein höherer NPS-Wert und höhere Suchtiefen erreicht werden konnten ist einleuchtend, da die Bewertungsfunktion Aufwand verursacht, der in diesem Test nicht geleistet werden mußte, es standen also mehr Ressourcen für die eigentliche Suche zur Verfügung. Im Vergleich zu dem Test mit Bewertungsfunktion wird deutlich, daß richtige Lösungen erst mit einer höheren Suchtiefe gefunden werden konnten, was zeigt, daß die Bewertungsfunktion in der Tat in der Lage ist, die Entwicklung einer Stellung früher zu erkennen, als über den reinen Materialwert möglich ist. Überraschend ist jedoch, daß trotz des Fehlens einer Bewertungsfunktion überhaupt 81% der möglichen Lösungen gefunden werden konnten: Dies liegt daran, daß Teststellungen eine definierte Lösung haben **müssen**, um als solche geeignet zu sein. Diese Lösung aber bestimmt sich zu einem großen Teil aus Materialgewinn oder -verlust, da diese Faktoren entscheidend für eine eindeutige Entwicklung einer Stellung sind. Da Teststellungen also diese Materialveränderung implizieren, sind sie nicht repräsentativ für das Schachspiel, wo es oft Situationen gibt, in denen keine eindeutige Materialentwicklung vorhanden ist. Es ist also nicht möglich, wie die obigen Resultate nahelegen würden, auf die Bewertungsfunktion zu verzichten, da sie in eben den Situationen unbedingt gebraucht wird, wo positive und negative Faktoren einer Stellung, die nicht durch Materialveränderung gekennzeichnet sind, erkannt und bewertet werden müssen.

Um zu zeigen, daß Stellungen vor allem deshalb nicht gelöst werden konnten, weil die rechnerischen Ressourcen des iPAQ H5500 dies nicht zulassen, habe ich Nemesis ebenfalls auf dem Pocket PC Emulator getestet. Hierfür habe ich einen AMD Athlon 2 Gigahertz PC mit 512 Megabyte RAM verwendet:

Lösungen Gefunden	94
Erfolgsquote	94%
Gesamtzeit(s)	311
Anzahl Knotenexpansionen	70752412
NPS	227499
Durchschnittliche Zeit(s) (gelöste Stellungen)	1.31
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	263106
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	7670062
Durchschnittliche Suchtiefe (gelöste Stellungen)	4.16
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	10.00

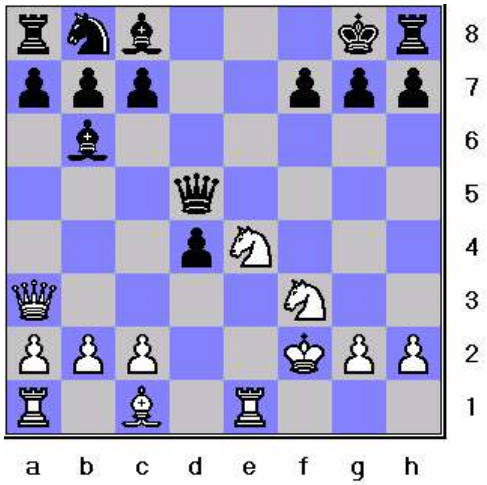
Abb. 5.4: Resultate von Teststellungen auf dem Emulator

Auf dem Emulator erreicht Nemesis 94% der möglichen Lösungen, wobei durchschnittlich nur 1,31 Sekunden benötigt werden, um die Lösung zu finden. Insgesamt können sehr viel höhere Suchtiefen erreicht werden, als auf dem iPAQ H5500, da der NPS-Wert etwa fünfmal so groß ist. Durch diese Daten wird deutlich, daß das Suchverfahren sowie die vorhandenen Optimierungen korrekt implementiert sind, da mehr Lösungen nicht aus algorithmischen Gründen, sondern nur aus Gründen des rechnerischen Aufwands nicht gefunden werden können. Die Anforderung 2.3, die ein korrektes Suchverhalten fordert, kann anhand dieser Ergebnisse als erfüllt betrachtet werden, da 94% gefundene Lösungen fehlerhafte Suchergebnisse ausschließen[28].

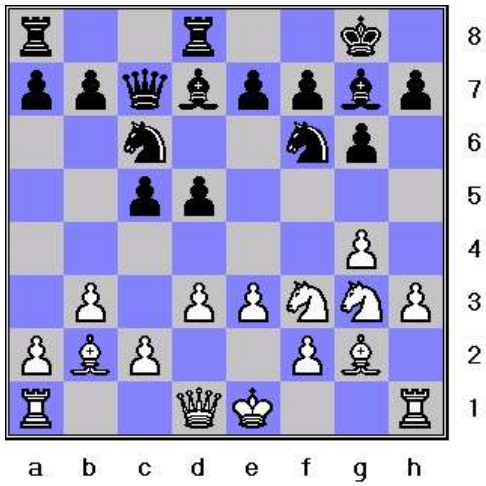
Ich werde in den weiteren Vergleichen der Optimierungsverfahren nur diejenigen in Betracht ziehen, die in Nemesis 1.0 verwendet wurden. Ein Vergleich der Alpha/Beta-Suche zur MinMax-Suche sowie des PVS/NegaScout-Algorithmus zur Alpha/Beta-Suche ist nicht notwendig, da die in den theoretischen Grundlagen dargestellte Überlegenheit dieser Verfahren bereits bewiesen wurde, z.B. in [Knuth; Moore 75 / Reinefeld 83 / Hartz 05].

5.3.2. Analyse von Beispielstellungen

Die WAC-Testsammlung besteht aus Stellungen, die sich fast ausschließlich im Mittel- und Endspielbereich befinden, da in der Eröffnung kaum eindeutig feststellbar sein kann, welcher der bestmögliche Zug ist[29]. Um Nemesis auch in der Eröffnung zu bewerten, verweise ich auf 5.4.2., wo ich Testspiele gegen TSCP 1.81 und GNUChess 4 analysiert habe. Ich habe einige Beispiele aus der Menge der Teststellungen herausgegriffen, um darzustellen, wie typische Stellungen aus der WAC-Sammlung aussehen:



Stellung: WAC139
Typ: Mittelspiel
bester Zug: e4f6

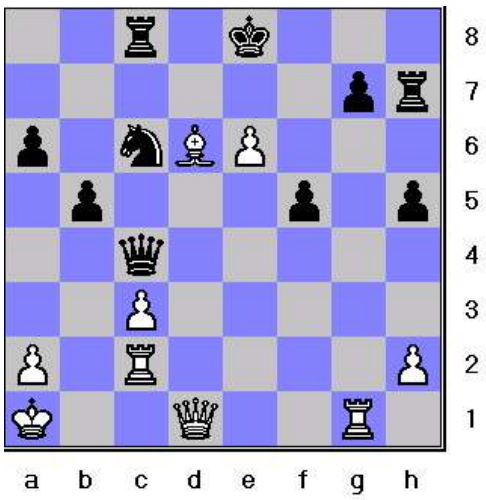


Stellung: WAC147
Typ: Mittelspiel
bester Zug: f6g4

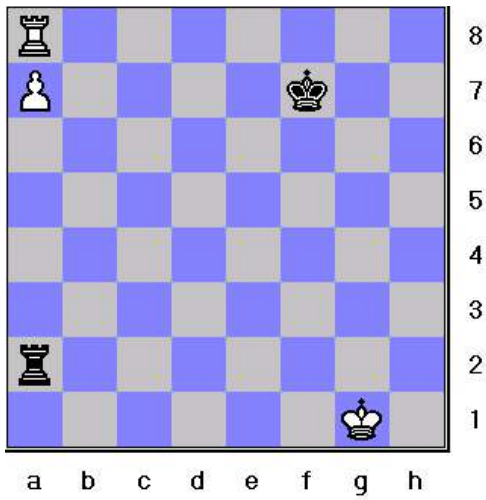
Abb. 5.5: Beispiele für Teststellung aus der WAC - Sammlung

28 Dieses Ergebnis wurde zwar nur auf dem Emulator erreicht, der algorithmische Ablauf ist aber der gleiche, wie auf der Zielplattform

29 Außer die Lösung ist trivial



Stellung: WAC148
Typ: Endspiel
bester Zug: g1g7



Stellung: WAC018
Typ: Endspiel
bester Zug: a8h8

Abb. 5.6: Beispiele für Teststellung aus der WAC - Sammlung

Viele der Stellungen aus der Testsammlung zeichnen sich durch ein Figurenopfer aus, das in den ersten Zügen stattfinden muß, um die richtige Lösung zu erreichen. Sie sind deshalb gut als Teststellungen geeignet, da ein Schachprogramm den Verlust einer Figur immer negativ bewertet, die Lösung also erst als besten Zug erkennt, wenn die daraus entstehende Lösungskombination zweifelsfrei erkannt wurde. Die zufällige Lösung einer Stellung ist dadurch ausgeschlossen.

Anhand dieser Beispielstellungen werde ich nun zeigen, welchen Einfluß verschiedene Verfahren auf die Anzahl der notwendigen Knotenexpansionen haben. Hierbei werde ich nur die Verfahren berücksichtigen, die einen großen Einfluß auf dieses Ergebnis gezeigt haben. Wenn im Text nicht anders erläutert, habe alle Tests zu den korrekten Lösungen geführt.
Die Stellungen unterscheiden sich deutlich in ihren Eigenschaften, was anhand der Resultate verdeutlicht, daß das Zusammenspiel der unterschiedlichen Verfahren einen entscheidenden Einfluß auf den Erfolg der Suche hat.

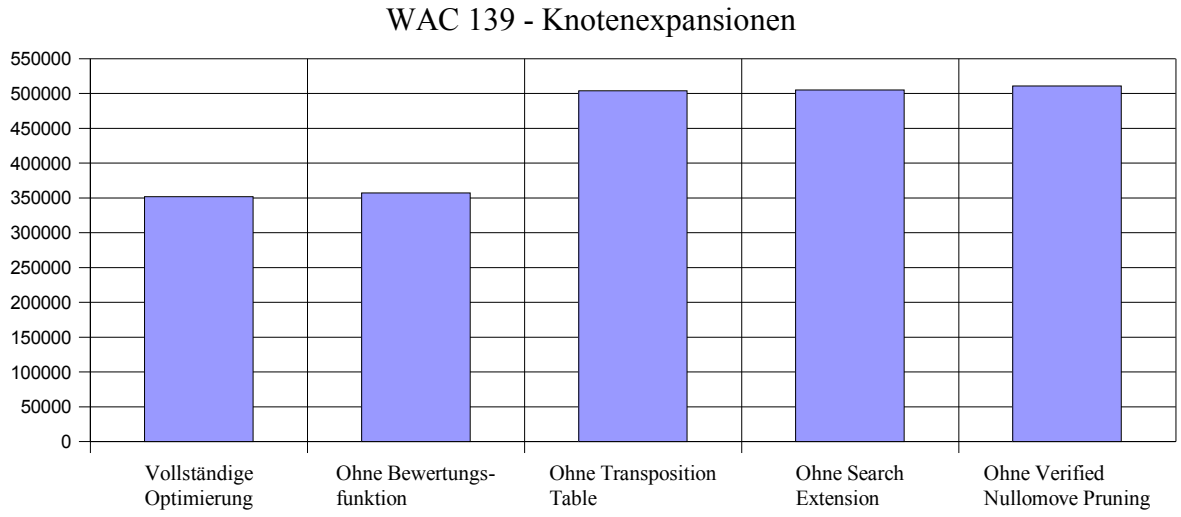


Abb. 5.7: Vergleich verschiedener Verfahren, Stellung WAC139

Die Stellung WAC139 stellt eine hohe Anforderung an die MinMax-Suche: Das korrekte Ergebnis kann bei vollständiger Optimierung erst in einer Suchtiefe von 6 Halbzügen (wobei noch maximal 8 Halbzüge Quiescence Search zusätzlich berücksichtigt werden müssen) gefunden werden, hierfür müssen 351680 Knoten untersucht werden. Bei deaktivierter Bewertungsfunktion kann das Suchergebnis in etwa genauso schnell gefunden werden, hierzu sind 357341 Knotenexpansionen notwendig. Sobald jedoch die Verwendung der Transposition Table, der Search Extensions oder des Verified Nullmove Pruning unterbunden wird, ist die MinMax-Suche trotz über 500000 Knotenexpansionen nicht mehr in der Lage, daß richtige Ergebnis zu finden. Dies zeigt sehr deutlich, daß diese Verfahren für ein effizientes Suchverhalten unerläßlich sind.

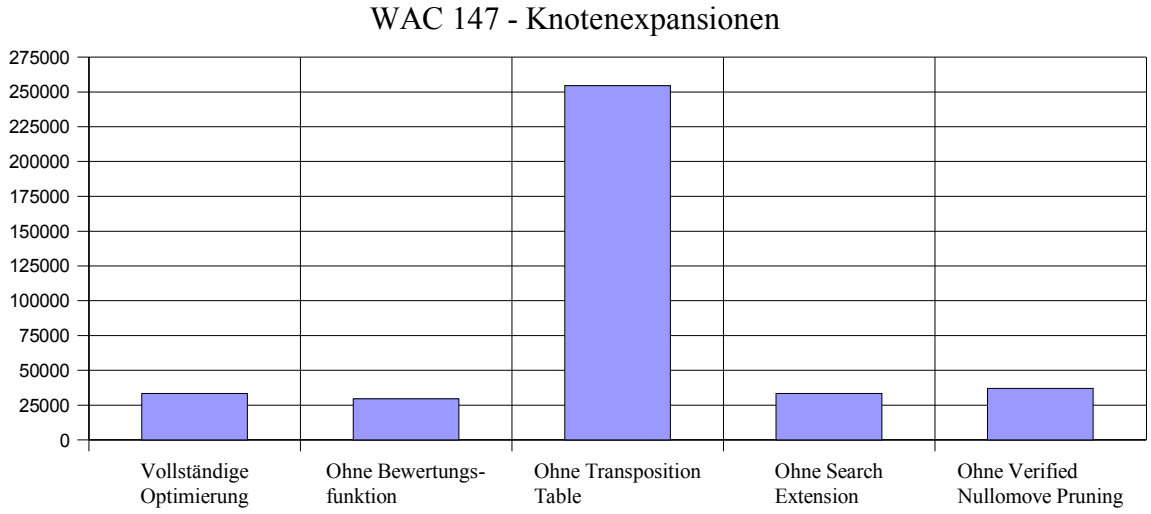


Abb. 5.8: Vergleich verschiedener Verfahren, Stellung WAC147

Bei der Analyse der Stellung WAC147 sind die Unterschiede nicht mehr ganz so eindeutig. Am auffälligsten ist, daß ohne Verwendung der Transposition Table fast zehnmal soviel Knoten benötigt werden, um die korrekte Lösung zu finden. Dies weist darauf hin, daß die Menge der notwendigerweise zu untersuchenden Knoten so groß ist, daß ohne Zwischenspeicherung von Resultaten ein deutlicher Overhead entsteht. Die Verwendung von Verified Nullmove Pruning hat keinen so deutlichen Einfluß, denn die Menge der notwendigen Knotenexpansionen steigt nur um etwa 10%, wenn man auf dieses Verfahren verzichtet. Search Extensions haben in dieser Stellung keine Bedeutung, was vermuten läßt, daß im Suchpfad zu der korrekten Lösung keine Bedingungen gegeben sind, die Search Extensions auslösen. Die Anzahl der Knotenexpansionen ist ohne Bewertungsfunktion etwas kleiner als mit, was darauf zurückzuführen ist, daß Materialentwicklungen in seltenen Fällen von der Bewertungsfunktion verdeckt werden können (s. 4.5.5.), was hier der Fall zu sein scheint.

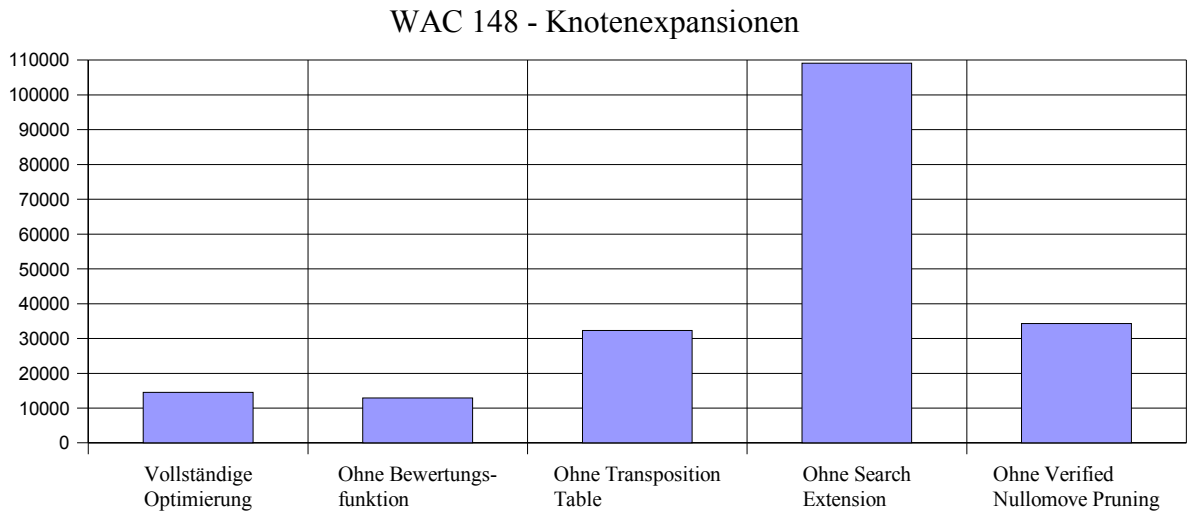


Abb. 5.9: Vergleich verschiedener Verfahren, Stellung WAC148

Die Anzahl der notwendigen Knotenexpansionen zeigt bei der Stellung WAC148 wieder deutlichere Unterschiede: Bei vollständiger Optimierung der Suche müssen 14535 Knoten untersucht werden, um die Lösung zu finden. Dieser Wert ist ohne Bewertungsfunktion etwas kleiner. Ohne Transposition Table steigt die Menge der Knotenexpansionen um etwa 150%, der Unterschied ist jedoch nicht so groß wie in der vorherigen Stellung, da bei WAC148 die Anzahl der nötigen Knotenexpansionen bei vollständiger Optimierung geringer ist. Ohne Verwendung von Verified Nullmove Pruning liegt sie in ähnlichen Bereichen. Besonders deutlich wird hier, welchen Einfluß Search Extensions haben können: Wenn sie nicht verwendet werden, so müssen für diese Stellung mehr als zehnmal soviel Knoten untersucht werden. Eine genauere Analyse der Stellung zeigt, daß hier sowohl die Check Extension als auch die Mate Threat Extension (s. 4.5.4.) dazu führen, daß die korrekte Lösung sehr viel früher gefunden werden kann. Dies ist deshalb gegeben, da der beste Zug g1g7 einen Opferangriff einleitet (... h7g7), der über eine Reihe von weiteren Schachgeboten der Dame (d1h5) zu einem Schachmatt führt. Die Schachgebote, sowie das drohende Matt führen dazu, daß Extensions ausgelöst werden, die es der Suche ermöglichen, die entscheidenden Ergebnisse deutlich früher zu finden, als es ohne Search Extensions möglich wäre.

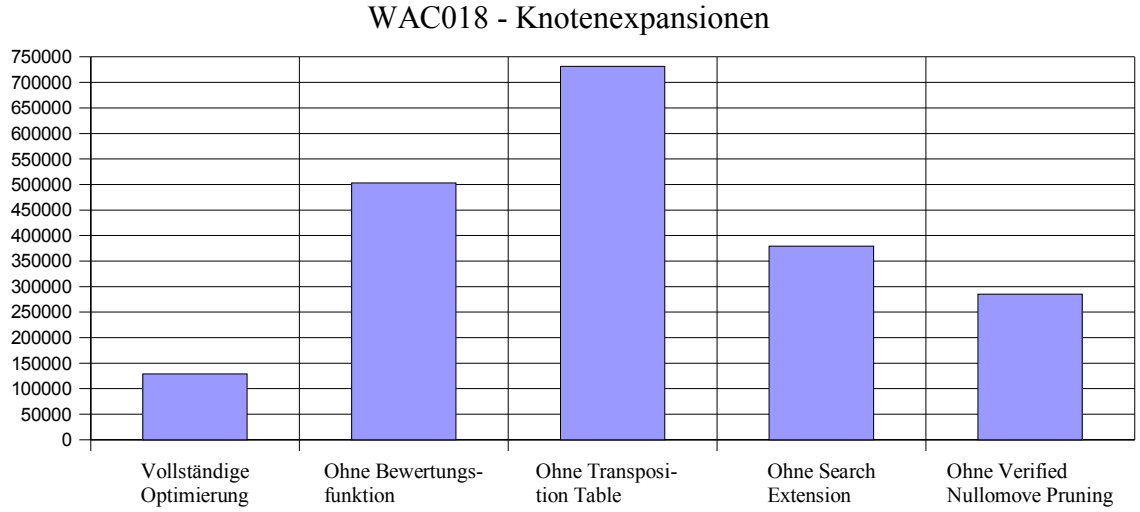


Abb. 5.10: Vergleich verschiedener Verfahren, Stellung WAC018

Die Stellung WAC018 führt ebenfalls zu einem deutlich unterschiedlichen Verhalten der verschiedenen Verfahren: Bei vollständiger Optimierung kann die Lösung in einer Suchtiefe von 8 Halbzügen gefunden werden, hierzu sind 128686 Knotenexpansionen notwendig. Ohne Verified Nullmove Pruning steigt diese Zahl bereits auf mehr als die doppelte Menge an. Search Extensions führen auch in dieser Stellung zu einer deutlichen Verbesserung der Effizienz, ohne Search Extensions verdreifacht sich die Anzahl der Knoten, die untersucht werden müssen, was vor allem auf die Pawn Promotion Extension und die Check Extension zurückzuführen ist. Ohne Bewertungsfunktion müssen 502916 Knoten analysiert werden, um die Lösung zu finden. Dies ist eine Steigerung um fast 400%, was zeigt, daß die Bewertungsfunktion in dieser Stellung sehr gut anspricht: Das Ziel des besten Zuges a8h8 ist es, den weißen Bauern in den nächsten Zügen umwandeln zu können, um nach dem folgenden Materialtausch mit dem verbleibenden Turm mattsetzen zu können. Dies wird von der Bewertungsfunktion antizipiert, da der Bauer kurz vor der Umwandlung bereits eine sehr hohe Bewertung erhält (s. 4.5.5.). Da zur Lösung dieser Stellung vergleichsweise viele Knoten analysiert werden müssen, spricht sie ebenfalls sehr gut auf die Verwendung der Transposition Table an, sollte diese deaktiviert werden, so müssen 731458 Knoten untersucht werden, dies entspricht mehr als dem fünffachen Wert dessen, was mit Zwischenspeicherung von Suchergebnissen notwendig ist.

Insgesamt wird aus diesen Beispielen deutlich, daß eine erfolgreiche Suche nicht von einem einzelnen Verfahren allein abhängen kann, da die zu untersuchenden Stellungen oft so unterschiedlich sind, daß sie auf gänzlich verschiedene Optimierungen ansprechen:
Eine gut Bewertungsfunktion ist wichtig für die Spielstärke (s. 5.4.3.), sie kann aber auch, wie bei WAC018 gut zu sehen ist, die Suche optimieren, indem sie die weitere Entwicklung der Stellung antizipiert und dadurch den richtigen Zug frühzeitig erkennt.
Die Verwendung einer Transposition Table ist essentiell für eine effiziente Suche, was aus allen Beispielen hervorgeht. Die Beschleunigung der Suche tritt vor allem dann stark hervor, wenn viele Knoten analysiert werden müssen, um die Lösung zu finden.
Search Extensions steigern die Qualität der Suchergebnisse, da wichtige Entwicklungen durch Extensions früher gefunden werden können. Dies trifft nicht in allen Stellungen zu, die Effizienzsteigerung ist abhängig von der gesuchten Zugabfolge. Den größten Effekt erzielen sie bei Bauernendspielen, wie an WAC018 zu erkennen ist, sowie bei Mattangriffen, die über mehrere Schachgebote ausgeführt werden, wie es bei WAC148 gegeben ist.
Mit Hilfe von Verified Nullmove Pruning konnte in allen Stellungen eine Beschleunigung der Suche erreicht werden, die deswegen unterschiedlich stark ausfällt, weil sie ebenfalls von den Charakteristika der Stellung abhängt. Gibt es in der Suche viele Zugzwang-Situationen, so fällt die Steigerung der Effizienz nicht so deutlich aus, dies ist bei WAC147 der Fall.

Eine detaillierte Darstellung der Auswirkungen aller getesteten Verfahren, bezogen auf die vollständige Menge der Teststellungen, findet sich in den nächsten Abschnitten. Dort werde ich darstellen, welche Auswirkungen auf die Resultate durch einzelne Verfahren entstehen, und wie diese zu begründen sind. Hierzu werde ich die Werte als Referenz benutzen, die mit der vollständigen Optimierung erreicht wurden.

5.3.3. Optimierung durch Vorsortierung

Daß die Vorsortierung der Knoten einen entscheidenden Einfluß auf die Effizienz der Suche hat, habe ich bereits in 2.1.5 erläutert. Wie signifikant dieser Einfluß ist, wird deutlich, wenn man die Suchergebnisse betrachtet, die ohne MVV/LVA- sowie Killer Heuristic - Vorsortierung erreicht werden:

Lösungen Gefunden	64
Erfolgsquote	64%
Gesamtzeit(s)	529
Anzahl Knotenexpansionen	23204023
NPS	43864
Durchschnittliche Zeit(s) (gelöste Stellungen)	4.14
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	75037
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	511157
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.13
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	4.19

Abb. 5.11: Resultate von Teststellungen ohne MVV/LVA- / Killer - Vorsortierung

Die Unterschiede zu den Resultaten mit vollständiger Vorsortierung sind erstaunlich: Die Menge der gefundenen Lösungen hat sich auf nur noch 64% der möglichen Stellungen reduziert, gleichzeitig ist die insgesamt benötigte Zeit um über 50% angestiegen. Die durchschnittliche Suchtiefe für korrekte Lösungen ist zwar gesunken, dies ist aber ein statistischer Effekt, da überhaupt nur einfache Stellungen, deren Lösung mit geringen Suchtiefen erreichbar ist, gelöst werden konnten. Die Suche hat sich insgesamt deutlich verlangsamt, was man auch daran erkennen kann, daß die durchschnittlich benötigte Zeit für gelöste Stellungen um ca. 20% angestiegen ist. Die durchschnittliche Suchtiefe für nicht gelöste Stellungen hat sich ebenfalls drastisch reduziert, was die Verlangsamung der Suche bestätigt.

Nicht ganz so gravierend fällt die Vorsortierung durch die Transposition Table ins Gewicht. Sie beschleunigt die Suche vor allem dadurch, daß durch die frühzeitige Erzeugung des besten Zuges mehr Schnitte im Suchbaum verursacht werden können. Dies führt zu einer leichten Erhöhung des NPS-Wertes, da die Suche an mehr Knoten mit einem Schnitt abgebrochen werden kann, also weniger Knoten vollständig untersucht werden müssen. Es muß auch berücksichtigt werden, daß die Transposition Table – Vorsortierung erst dann wirklich ins Gewicht fällt, wenn mehrere Züge hintereinander gespielt werden, weil Informationen über die Grenzen der Spielzüge gehalten werden können, und zu Beginn eines neuen Zuges zur Verfügung stehen. Dies gilt natürlich nicht für Teststellungen, da hier jede Suche an einer vollständig neuen Stellung begonnen werden muß, und die Daten der Transposition Table deswegen vor jeder neuen Stellung gelöscht werden.

Lösungen Gefunden	83
Erfolgsquote	83%
Gesamtzeit(s)	392
Anzahl Knotenexpansionen	15323521
NPS	39090
Durchschnittliche Zeit(s) (gelöste Stellungen)	3.46
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	66557
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	576425
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.71
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	5.88

Abb. 5.12: Resultate von Teststellungen ohne Transposition Table - Vorsortierung

Ein Verfahren, welches ich ebenfalls verwendet habe, sind Search Extensions. Sie dienen zwar nicht direkt dazu, die Vorsortierung zu verbessern, die theoretische Grundannahme ist jedoch ähnlich: Auch Search Extensions sollen dafür sorgen, daß wichtige Ergebnisse frühzeitig gefunden werden, um den Ablauf der Suche zu beschleunigen sowie die Qualität der Ergebnisse zu verbessern.

Lösungen Gefunden	80
Erfolgsquote	80%
Gesamtzeit(s)	437
Anzahl Knotenexpansionen	16309492
NPS	37321
Durchschnittliche Zeit(s) (gelöste Stellungen)	3.04
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	57244
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	586497
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.85
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	7.10

Abb. 5.13: Resultate von Teststellungen ohne Search Extensions

Man erkennt, daß Search Extensions tatsächlich einen positiven Einfluß auf die Menge der gefundenen Lösungen haben, da sich deren Anzahl gegenüber der vollständigen Version von Nemesis ohne Search Extensions um 5% reduziert hat. Obwohl weniger (also einfachere) Lösungen gefunden wurden, hat sich die hierfür benötigte Suchtiefe erhöht. Search Extensions sind also sinnvoll, um die Suche auf relevante Ergebnisse zu fokussieren. Gleichzeitig erkennt man auch, daß sie einen zusätzlichen Aufwand erzeugen, denn die durchschnittliche Suchtiefe für nicht gelöste Stellungen ohne Search Extensions ist deutlich höher. Dies ist einleuchtend, da bei der Verwendung von Search Extensions die Suchtiefe im Ablauf der Suche selektiv erhöht wird, was zu mehr Knotenexpansionen, also größerem Aufwand, führt.

5.3.4. Optimierung der Alpha/Beta-Suche

Grundlegend für die Optimierung einer Alpha/Beta-Suche ist die Verwendung einer Transposition Table. Die Zwischenspeicherung von Ergebnissen führt zu einer deutlichen Beschleunigung der Suche. Hierbei muß, wie oben erläutert wurde, berücksichtigt werden, daß diese Beschleunigung in einer Spielanwendung noch wesentlich deutlicher ausfällt: Hier können mehrere Suchen hintereinander durch vorhandene Informationen der Transposition Table beschleunigt werden, wohingegen bei Teststellungen für jede Suche die Daten der Transposition Table gelöscht werden müssen. Die Resultate eines Testdurchlaufs ohne Transposition Table sind wie folgt:

Lösungen Gefunden	81
Erfolgsquote	81%
Gesamtzeit(s)	497
Anzahl Knotenexpansionen	17681346
NPS	35576
Durchschnittliche Zeit(s) (gelöste Stellungen)	4.02
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	78134

Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	597499
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.65
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	6.11

Abb. 5.14: Resultate von Teststellungen ohne Transposition Table

Am deutlichsten fällt bei diesen Testresultaten auf, daß sich die insgesamt benötigte Zeit um ca. 50% erhöht hat, was einen enormen Zuwachs gegenüber der Version mit Transposition Table darstellt. Die Anzahl der notwendigen Knotenexpansionen zur Findung von Lösungen hat sich ebenfalls erhöht, genauso wie die für gelöste Stellungen notwendige Zeit. Dies verdeutlicht, wie wichtig die Speicherung von Suchergebnissen ist, gerade wenn diese eine Ergebnissveränderung darstellen. Der NPS-Wert hingegen ist stark gesunken, was darauf zurückzuführen ist, daß deutlich weniger Alpha/Beta-Schnitte im Ablauf der Suche möglich waren. Dies beweist den positiven Effekt der Transposition Table insbesondere auf diesen Faktor: Die Transposition Table hat den größten Einfluß auf Alpha/Beta-Verfahren in der Art, daß sie nicht nur Ergebnisse vorheriger Knoten speichert, um Wiederholungssuchen zu vermeiden, sondern auch, um mehr Schnitte im Suchbaum zu ermöglichen, falls bereits eine Ober- oder Untergrenze zu einem untersuchten Knoten bekannt ist. Diese Art der Optimierung bezieht sich ausschließlich auf Alpha/Beta-Verfahren, wohingegen vom allgemeinen Vorhandensein einer Transposition Table auch andere Suchverfahren profitieren können.

Eine weitere Optimierung der Alpha/Beta-Suche liegt in der Verwendung der Aspiration Search, da diese durch eine Verkleinerung des initialen Suchfensters ebenfalls versucht, die Menge der im Suchbaum möglichen Schnitte zu erhöhen:

Lösungen Gefunden	82
Erfolgsquote	82%
Gesamtzeit(s)	361
Anzahl Knotenexpansionen	14658968
NPS	40606
Durchschnittliche Zeit(s) (gelöste Stellungen)	2.77
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	50507
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	584298
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.67
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	6.28

Abb. 5.15: Resultate von Teststellungen ohne Aspiration Search

Der Unterschied gegenüber der vollständigen Optimierung ist deutlich erkennbar: Der NPS-Wert ist gesunken, was zeigt, daß weniger Alpha/Beta-Schnitte im Suchbaum möglich waren. Die Anzahl der gefundenen Lösungen ist ebenfalls gesunken, was natürlich auch zu einem Ansteigen der Gesamtzeit führt. Die durchschnittliche Zeit für gefundene Lösungen konnte aber verringert werden, was einen spezifischen Effekt der Aspiration Search verdeutlicht: Weil Teststellungen Materialveränderungen implizieren (s. 5.3.1.), sind bei Anwendung der Aspiration Search viele Wiederholungssuchen an der Wurzel notwendig, da die Ergebnisse zur Lösung einer Stellung fast immer einen Gewinn darstellen, der das Aspiration Window verletzt. Wenn z.B. irgendwann der Gewinn eines Bauern erkannt wird, so liegt dieser Wert schon außerhalb des Aspiration Window, das dem Ergebnis der vorherigen Suchtiefe +/- dem Drittel des Wertes eines Bauern entspricht.

5.3.5. Weitere Optimierungen

Verified Nullmove Pruning ist die bisher einzige betrachtete Optimierungsmaßnahme, die nur für eine Schachanwendung tauglich ist, da die grundlegende Heuristik auf dem spezifischen Ablauf des Schachspiels basiert (s. 2.2.8.). Dieses Verfahren ist gleichzeitig jedoch äußerst wichtig für die Effizienz der MinMax-Suche einer Schachanwendung, wie die Testresultate zeigen, die ohne Verified Nullmove Pruning erreicht wurden:

Lösungen Gefunden	79
Erfolgsquote	79%
Gesamtzeit(s)	434
Anzahl Knotenexpansionen	15883136
NPS	36597
Durchschnittliche Zeit(s) (gelöste Stellungen)	2.66
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	48819
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	572685
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.52
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	5.62

Abb. 5.16: Resultate von Teststellungen ohne Verified Nullmove Pruning

Der NPS-Wert hat sich um ca. 15% verringert, was dazu führt, daß 7% weniger Lösungen gefunden werden können. Die durchschnittliche Suchtiefe für nicht gelöste Stellungen ist ebenfalls sehr deutlich gesunken, und zwar um 17%, was den positiven Effekt des Verified Nullmove Pruning auf die Geschwindigkeit der Suche eindrucksvoll bestätigt.

Zwei weitere Verfahren, die spezifisch für die MinMax-Suche in einer Schachanwendung sind, sind Futility Pruning und Lazy Eval. Diese beiden Verfahren können nur verwendet werden, da eine Quiescence Search implementiert ist (s. 2.2.9. und 2.2.10.).

Lösungen Gefunden	83
Erfolgsquote	83%
Gesamtzeit(s)	383
Anzahl Knotenexpansionen	15297641
NPS	39941
Durchschnittliche Zeit(s) (gelöste Stellungen)	3.41
Durchschnittliche Knotenexpansionen (gelöste Stellungen)	63357
Durchschnittliche Knotenexpansionen (nicht gelöste Stellungen)	590525
Durchschnittliche Suchtiefe (gelöste Stellungen)	3.71
Durchschnittliche Suchtiefe (nicht gelöste Stellungen)	6.35

Abb. 5.17: Resultate von Teststellungen ohne Futility Pruning und ohne Lazy Eval

Man erkennt, daß diese beiden Verfahren nur einen leichten effizienzsteigernden Einfluß haben. Ohne ihre Verwendung sinkt der NPS-Wert leicht und es konnten zwei Lösungen weniger gefunden werden. Auch die durchschnittliche Suchtiefe für nicht gelöste Stellungen hat sich etwas verringert.

5.3.6. Vergleich der Optimierungen

Ich möchte nun einen abschließenden Vergleich anstellen, um zu zeigen, wie sich die Anzahl der gefundenen Lösungen durch die Verwendung verschiedener Optimierungsverfahren geändert hat:

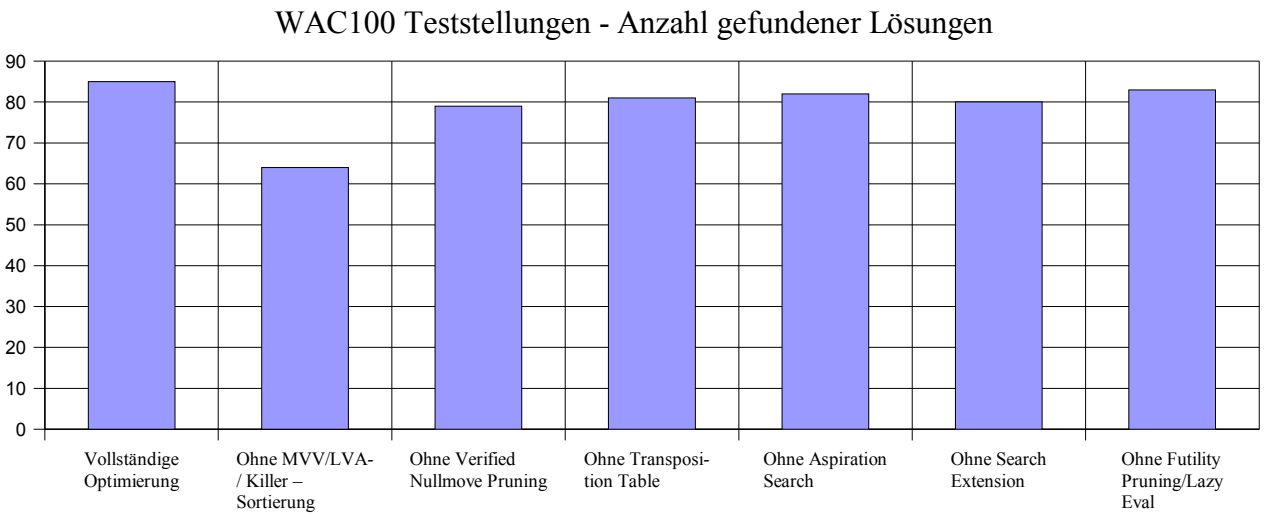


Abb. 5.18: Anzahl gefundener Lösungen anhand verschiedener Optimierungen

Anhand dieser Grafik wird deutlich, was das Besondere bei der Optimierung einer MinMax-Suche ist: Wenn ein schneller Suchalgorithmus verwendet wird, wie der PVS/NegaScout-Algorithmus in diesem Fall, so ist es schwierig, ihn weiter zu optimieren. Wichtig hierfür ist die Vorsortierung der Knoten, was sehr deutlich wird, wenn man die Testergebnisse ohne MVV/LVA / Killer - Vorsortierung betrachtet. Die Verwendung einer Transposition Table ist ebenfalls essentiell, was durch Teststellungen jedoch nur bedingt gezeigt werden kann. Außerdem sind die Ressourcen des verwendeten PDA in diesem Bereich zu gering, um den positiven Effekt deutlicher darzustellen. Weitere Optimierungen sind noch zu erreichen, allerdings nicht mehr durch einen einzigen Ansatz: Jedes der weiteren gezeigten Verfahren steigert die Effizienz, da erst bei vollständiger Optimierung die größte Anzahl von Lösungen gefunden werden konnte. Dieser Effekt wirkt bei vielen der Optimierungen marginal, in der Summe erreichen sie jedoch einen deutlichen Effizienzzuwachs. Dieser Zuwachs hängt maßgeblich von den verschiedenen **Charakteristika** einer zu untersuchenden Stellung ab, was in 5.3.2. bereits anhand einiger Beispiele gezeigt wurde. Die spezifischen Eigenschaften einer Stellung umfassen hierbei so unterschiedliche Kriterien wie die Menge des vorhandenen Materials, mögliche Schlagkombinationen, mögliche Schachgebote, Patt-Situationen, Zugzwang-Situationen, Auslösungen von Search Extensions usw... . Um eine Schachanwendung in ihrer Spielstärke zu optimieren, müssen alle diese Kriterien gleichermaßen beachtet werden, da nur so garantiert ist, daß das Programm in jeder möglichen Spielsituation eine gute Leistung erbringen kann. Entscheidend für die Optimierung einer MinMax-Suche ist also das **Zusammenwirken** der verschiedenen Verfahren.

Man erkennt in der Analyse der verschiedenen Optimierungsmaßnahmen die aus dem Software Engineering bekannte Pareto-Verteilung[30], die besagt, daß in bestimmten Systemen „eine kleine Anzahl von bewerteten Elementen in einer Menge sehr viel zum Gesamtwert der Menge beitragen, wohingegen der überwiegende Teil der Elemente nur sehr wenig zum Gesamtwert beiträgt“ [Wikipedia Pareto]. Diese Verteilung der unterschiedlichen Signifikanz von Elementen ist auch in der Optimierung der MinMax-Suche vorhanden: Ein großer Teil der möglichen Performanz kann

30 Auch als 80/20 – Regel bekannt

durch die Verwendung des PVS/NegaScout-Algorithmus mit MVV/LVA- / Killer – Vorsortierung und Transposition Table erreicht werden. Die restlichen Effizienzsteigerungen, die noch möglich sind, entfallen auf alle weiteren Verfahren wie Verified Nullmove Pruning, Iterative Deepening, Aspiration Search, Search Extensions, Futility Pruning und Lazy Eval. Diese Verfahren leisten einen zusätzlichen Beitrag zur Gesamtperformanz, der jedoch entscheidend von der untersuchten Stellung abhängt (s.o.).

5.3.7. Leistungsfähigkeit der Zielplattform

In 5.3.1. wurde bereits ein Vergleich der Testresultate auf der Zielplattform und auf dem Emulator durchgeführt. Dieser Vergleich hat das Ergebnis geliefert, daß auf dem Emulator etwa ein fünffaches der Leistungsfähigkeit des iPAQ H5500 erreicht wurde. Dieses Ergebnis war zu erwarten, da der Emulator die Zielumgebung simulierte und auf einem System lief, dessen Systemtakt fünfmal so groß ist wie der der Zielplattform. Interessant ist nun ein Vergleich zwischen diesen Ergebnissen und den Ergebnissen einer Version von Nemesis auf dem PC, die nicht auf dem Emulator lief. Diese Version entspricht exakt der Version auf der Zielplattform[31], allerdings wurde sie speziell für den PC kompiliert.

Lösungen Gefunden	94
Erfolgsquote	94%
Gesamtzeit(s)	288
Anzahl Knotenexpansionen	84629042
NPS	293850
Durchschnittliche Zeit(s) (Gelöste Stellungen)	1.06
Durchschnittliche Knotenexpansionen (Gelöste Stellungen)	262773
Durchschnittliche Knotenexpansionen (Nicht Gelöste Stellungen)	9988053
Durchschnittliche Suchtiefe (Gelöste Stellungen)	4.16
Durchschnittliche Suchtiefe (Nicht Gelöste Stellungen)	10.83

Abb. 5.19: Resultate von Teststellungen auf dem PC

Man erkennt, daß die PC-Version von Nemesis die besten erreichten Resultate erzielt, was durch die geringste Gesamtzeit deutlich wird. Der NPS-Wert ist fast siebenmal so hoch wie auf dem iPAQ H5500. Dies zeigt, daß die Einschränkungen dieser Plattform nicht nur in einem geringeren Systemtakt liegen, der diesen Unterschied nicht rechtfertigen kann (s.o.). An einem Vergleich zu den Emulator-Ergebnissen wird ebenfalls deutlich, daß die Zielumgebung speziellen Overhead verursacht: Der NPS-Wert ist hier fast 30% geringer als ohne Emulator. Natürlich verursacht der Emulator eigenen Overhead, dieser liegt jedoch nur bei etwa 3% der im Test verbrauchten CPU-Zeit. Es gibt also einen deutlichen Unterschied in den Testergebnissen auf einem PC, der ausschließlich durch die simulierte Zielumgebung entsteht. Diese Ergebnisse verdeutlichen, daß die Zielplattform die Ausführungsgeschwindigkeit auf zwei Arten beeinflußt: Erstens durch ihre offensichtliche Beschränkung im Systemtakt, wie der Vergleich der Resultate der Zielplattform und des Emulators verdeutlicht. Außerdem gibt es jedoch auf der Zielplattform weitere Performanzverluste gegenüber der Version auf dem PC, die nicht durch technische Daten erklärbar sind. Eine Möglichkeit für diese Verluste wäre ein zu langsamer interner Datenbus, Informationen hierüber sind jedoch leider aus den technischen Beschreibungen des HP iPAQ H5500 [Hewlett Packard] nicht zu entnehmen.

31 Auch in der Größe der Transposition Table

5.4. Ergebnisse von Testspielen

5.4.1. Resultate der Testspiele

Zur Bewertung der Spielstärke von Nemesis 1.0 habe ich insgesamt acht Testspiele durchgeführt. Diese Anzahl ist für eine empirische Analyse der Anwendung nicht ausreichend, sie kann jedoch einen ungefähren Eindruck von der Spielstärke des Programms vermitteln. Nemesis hat vier Spiele gegen jeden der beiden Gegner gemacht, wobei alternierend mit den weißen und schwarzen Figuren gespielt wurde. Die Spiele wurden nach den üblichen Turnierregeln ausgetragen, jedes Programm hatte pro Zug 30 Sekunden Zeit, also insgesamt 2 Stunden für 40 Züge. TSCP 1.81 und GNUChess 4 spielten auf einem AMD Athlon 2 Gigahertz PC mit 512 Megabyte RAM, der natürlich deutlich höhere Rechenleistungen erreicht, als der HP iPAQ H5500 PDA. Eine Darstellung aller Spiele in algebraischer Notation (s.u.) findet sich in Anhang C „Testspiele“, sowie auf der beiliegenden CD.

Die einzelnen Testspiele haben zu folgenden Resultaten geführt[32]:

Programm	**Gesamt**	**Spiel 1**	**Spiel 2**	**Spiel 3**	**Spiel 4**
Nemesis 1.0	3,5	1	1	1	0,5
TSCP 1.81	0,5	0	0	0	0,5

Programm	**Gesamt**	**Spiel 1**	**Spiel 2**	**Spiel 3**	**Spiel 4**
Nemesis 1.0	0	0	0	0	0
GNUChess 4	4	1	1	1	1

Abb. 5.20: Resultate von Testspiele

Im Vergleich Nemesis 1.0 zu TSCP 1.81 hat sich gezeigt, daß beide Programme ungefähr die gleichen Suchtiefen erreichen, was aufgrund der unterschiedlichen Plattformen implementationsspezifische Gründe haben muß. Insgesamt haben mich die deutlichen Resultate der Testspiele gegen diesen Gegner überrascht: Ich hatte gehofft, daß Nemesis etwa die gleiche Spielstärke wie TSCP erreichen würde, daß jedoch ein deutlicher Sieg möglich ist, hatte ich nicht erwartet. Nach den erfolgreichen Resultaten der beiden ersten Spiele habe ich für die letzten beiden Spiele die Eröffnungsbibliothek von Nemesis deaktiviert, um zu testen, wie sich die Spielweise ohne diese Unterstützung verändert. Spiel 3 wurde von Nemesis mit den weißen Figuren bestritten und konnte in 58 Zügen gewonnen werden (s.u.), Spiel 4 mit den schwarzen Figuren führte zu einem Patt durch Wiederholung[33].

Das Nemesis gegen GNUChess 4 einen schweren Stand haben würde, hatte ich erwartet. Dieses Programm erreicht durchschnittlich etwa zwei bis drei Halbzüge Suchtiefe mehr als Nemesis, was einen extremen Vorteil darstellt. Nemesis hat alle Spiele gegen diesen Gegner verloren, wobei GNUChess durchschnittlich 57 Züge gebraucht hat, um zu gewinnen. Dies zeigt zumindest, daß Nemesis eine gewisse Spielzeit Widerstand leisten konnte, bevor eine Partie verloren wurde.

32 1 - Partie gewonnen , 0 - Partie verloren, 0,5 - Remis

33 Nemesis hatte einen Bauern mehr im Endspiel, dann wurde ein „unendliches Schach“ erzeugt: Eine Seite führt immer wieder ein Schachgebot aus, da alle Zugalternativen schlechter sind, und die andere Seite antwortet immer mit dem selben Zug, da alle anderen Möglichkeiten zu einem deutlichen Verlust führen [Surratt]

Ich werde im nächsten Abschnitt zwei der Testspiele detailliert beschreiben, um zu zeigen, wie Nemesis gegen die verschiedenen Gegner gespielt hat. Ich habe eine Partie gegen TSCP und eine gegen GNUChess ausgewählt, die exemplarisch die unterschiedlichen Spielstärken aufzeigen. Die Analyse der Spielabläufe werde ich später verwenden, um zu erläutern, welche Unterschiede zwischen den Programmen zu den unterschiedlichen Abläufen und Resultaten der Testspiele geführt haben. Hierzu habe ich sowohl die erreichten Suchtiefen, als auch die Suchergebnisse der beteiligten Programme protokolliert.

5.4.2. Exemplarische Darstellung und Analyse

Die Darstellung der Partien erfolgt in der hierfür üblichen algebraischen Notation [Wikipedia Notation]. Die Großbuchstaben stehen für eine Figur einer bestimmten Kategorie, wobei englische Bezeichnungen verwendet werden: N – Knight (Springer), B – Läufer (Bishop), Q – Dame (Queen), K – König (King). Ist keine Figur angegeben, so handelt es sich um einen Bauernzug. Sollte ein Zug für mehrere Figuren möglich sein, so wird die Startlinie der Figur zusätzlich angegeben. Das Zeichen 'x' repräsentiert einen Schlagzug, '+' ein Schachgebot und '#' ein Schachmatt. Nach den angegebenen Zeichen folgt das Zielfeld des Zuges. 'O-O' bezeichnet die Rochade auf dem Königsflügel, 'O-O-O' die Rochade auf dem Damenflügel.

Die erste Partie, die ich vorstellen möchte, ist Spiel 3 zwischen Nemesis 1.0 und TSCP 1.81. Diese Partie wurde von Nemesis ohne Eröffnungsbibliothek gespielt und zeigt, wie Nemesis diesen Spielabschnitt ohne Unterstützung bestreitet..

Weiß: Nemesis 1.0 – Schwarz: TSCP 1.81 (Spiel 3)

1. e4 Nc6 2. d4 e6

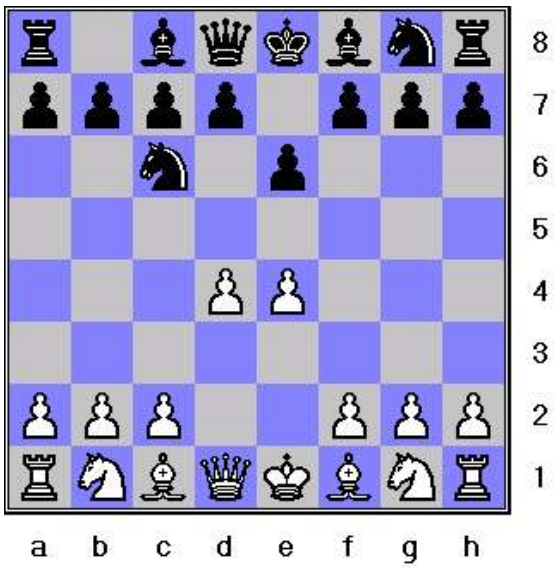


Abb. 5.21: 2. Zug, Eröffnung (Nemesis 1.0 – TSCP 1.81)

Weiß kontrolliert das Zentrum mit Bauern, gleichzeitig werden die Diagonalen für beide Läufer geöffnet. Schwarz kontert mit dem Springer b8 und öffnet ebenfalls eine Diagonale.

3. Bb5 Qh4

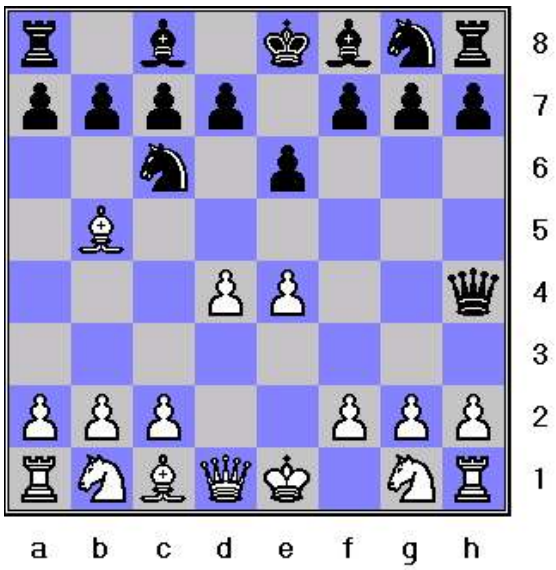


Abb. 5.22: 3. Zug, Verlauf der Eröffnung (Nemesis 1.0 – TSCP 1.81)

Weiß spielt in solchen Situationen oft f1b5, da dieser Zug den Springer c6 angreift, und bei einem Abtausch zu einer Verdopplung des Bauern d7 führt. Schwarz versucht mit d8h4 die starke Bauernfront des Gegners unter Druck zu setzen.

4. Bxc6 dxc6 5. Qe2 Bd7 6. Nf3 Qg4 7. g3 O-O-O 8. O-O c5

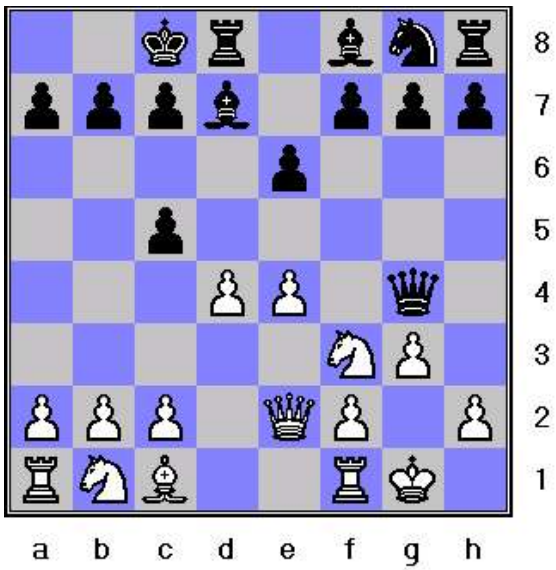


Abb. 5.23: 8. Zug, Verlauf der Eröffnung (Nemesis 1.0 – TSCP 1.81)

Beide Seiten haben eine Rochade ausgeführt, um ihre Stellung zu festigen. Man erkennt, daß es Nemesis auch ohne Eröffnungsbibliothek gelungen ist, eine gute Position für das Mittelspiel zu erreichen: Das Bauernzentrum ist stark, der Springer f3 übt hier zusätzliche Kontrolle aus. Der Läufer c1 hat eine offene Diagonale und kann gut ins Spiel geführt zu werden, die Dame greift von e2 aus bereits die Königsstellung des Gegners an. Schwarz hat sich ebenfalls entwickelt, wobei der Doppelbauer auf c5 ein Problem darstellt, und auch der Läufer auf d7 steht nicht optimal, da er nur wenig gute Zugmöglichkeiten hat.

9. h3 Bb5 10. Qxb5 Qxf3 11. Nd2 Qh5 12. Nb3 a6 13. Qa5 Nf6 14. Bf4 Rd7 15. dxc5 c6

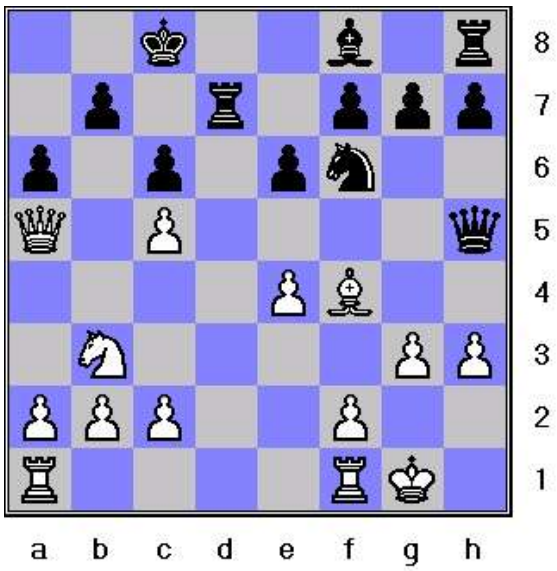


Abb. 5.24: 15. Zug, Übergang ins Mittelspiel (Nemesis 1.0 – TSCP 1.81)

Weiß hat die Dame nun aktiv ins Spiel geführt, von a5 aus bedroht sie massiv den gegnerischen König. Unterstützt wird sie vom Läufer f4, der ebenfalls die gegnerische Königsstellung attackiert. Der Springer b1 konnte von Weiß ins Spiel gebracht werden, so daß die Stellung insgesamt gut ausgebaut ist. Schwarz steht ebenfalls nicht schlecht, die Dame h5 übt Druck auf die weiße Königsdeckung aus. Der Springer f6 und der Turm d7 sind stark postiert, da sie das Zentrum des Spielfeldes angreifen können. Der Läufer f1 ist noch nicht im Spiel, dadurch blockiert er die Entwicklung des zweiten Turms.

16. e5 Nd5 17. Bc1 Qxe5 18. Re1 Qf5 19. c4 Nf6 20. Kg2 Be7 21. Bf4 Ne4 22. Rad1 Rhd8

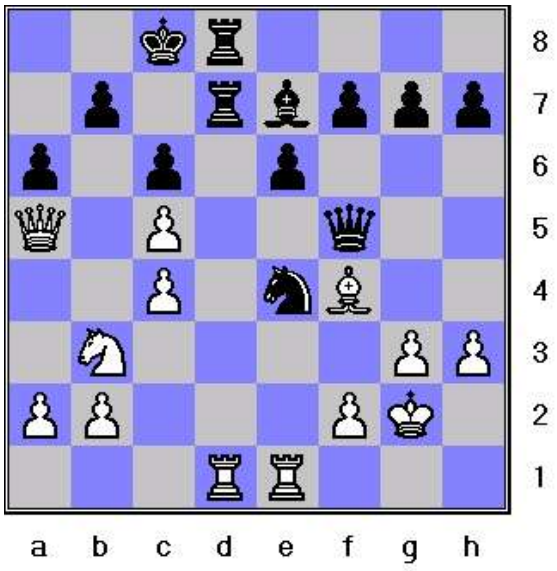


Abb. 5.25: 22. Zug, Verlauf des Mittelspiels (Nemesis 1.0 – TSCP 1.81)

Durch einen Bauerntausch hat Weiß nun einen Doppelbauer. Dies wurde von Nemesis in Kauf genommen, da so die d- und e-Linie für die Türme geöffnet werden konnten. Schwarz hat ebenfalls

seine Türme ins Spiel gebracht, leidet jedoch immer noch unter den offenen Angriffsdiagonalen der weißen Dame und des weißen Läufers.

23. Rxd7 Rxd7 24. Be3 e5 25. g4 Qe6 26. f3 Qxc4 27. fxe4 Qxe4+ 28. Kg1 Qf3 29. Bf2 Qxh3 30. Rxe5 Qxg4+

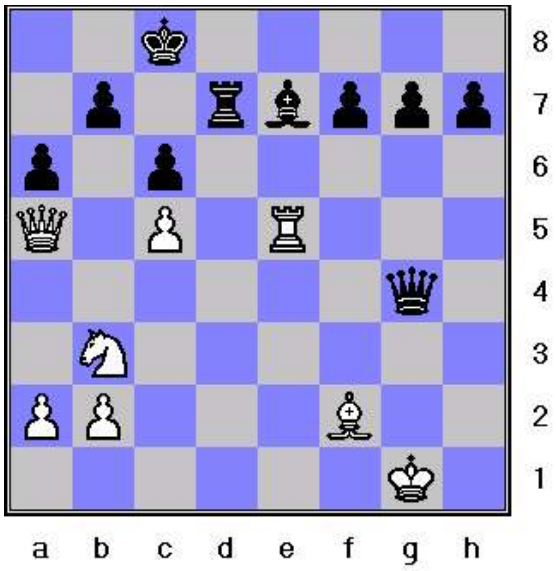


Abb. 5.26: 30. Zug, Verlauf des Mittelspiels (Nemesis 1.0 – TSCP 1.81)

Die Stellung hat sich deutlich verändert: Weiß konnte die starke Positionierung der Dame und des Läufers erhalten, ein Turm konnte ebenfalls sehr stark auf e5 positioniert werden. Nemesis hat eine Überlegenheit an Leichtfiguren erreicht, hierfür jedoch die komplette Königsdeckung aufgeben müssen: Dies ist zwar riskant, ein möglicher Damenangriff über b6, mit Unterstützung des Turms und des Springers, scheint dieses Risiko jedoch wert zu sein. Schwarz hat immer noch eine starke Dame auf dem gegnerischen Königsflügel, ansonsten konnte keine weitere Entwicklung erreicht werden. Da der Turm d7 zur Verteidigung eigener Bauern gefesselt ist, stehen TSCP kaum Angriffskombinationen zur Verfügung, um die fehlende Königsdeckung von Nemesis auszunutzen.

31. Kf1 f5 32. Re1 g5 33. Qb6 Qc4+ 34. Kg1 h5 35. Qa7 Qg4+ 36. Kh2 h4

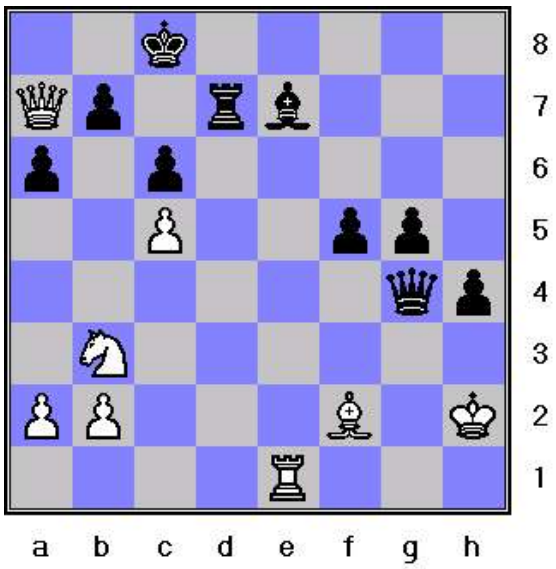


Abb. 5.27: 36. Zug, Verlauf des Mittelspiels (Nemesis 1.0 – TSCP 1.81)

Die Entscheidung steht bevor: Weiß konnte mit der Dame in die gegnerische Stellung eindringen und bereitet einen Angriff auf a8 vor. Die Gefährlichkeit dieses Angriffs ist nicht zu unterschätzen, da er sowohl vom Springer b3 als auch vom Läufer f2 unterstützt werden kann. Schwarz hat seine Bauern auf dem gegnerischen Königsflügel entwickelt und droht mit einem Angriff der Dame und möglichen Umwandlungen der Bauern.

37. Qa8+ Kc7 38. Nd4 Bxc5 39. Ne6+ Kb6 40. Nxc5 Qf4+ 41. Kg2 h3+ 42. Kf1 Qc4+ 43. Kg1 h2+

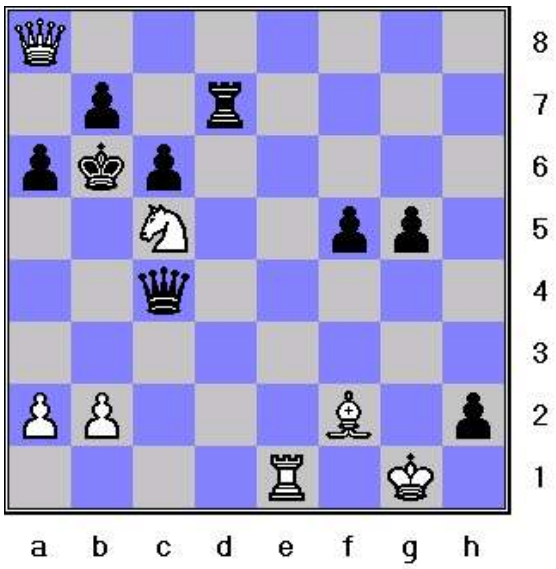


Abb. 5.28: 43. Zug, Verlauf des Mittelspiels (Nemesis 1.0 – TSCP 1.81)

Der weiße Angriff über a8 ist erfolgt und der schwarze Läufer konnte durch weitere Schachgebote des Springers erobert werden. Die Übermacht an Leichtfiguren stellt sich nun als deutlicher Vorteil heraus: Der weiße Läufer f2 ist stark postiert, da durch den Springer ein Abzugsschach[34] möglich ist. Schwarz greift immer noch mit der Dame an, außerdem konnte der h-Bauer bis ein Feld vor die

34 Der Gegner wird Schach gestellt, da durch den Abzug einer Figur ein Schach durch eine andere Figur entsteht

Umwandlung geführt werden. Es gibt für Schwarz jedoch keine weiteren Möglichkeiten, die Stellung positiv zu verändern, da die weißen Figuren durch viele mögliche Schachgebote zu großen Druck ausüben.

44. Kh1 Qd5+ 45. Ne4+ Ka5 46. Qh8 Kb5 47. Qc3 b6 48. a4+ Kxa4 49. Bxb6 c5

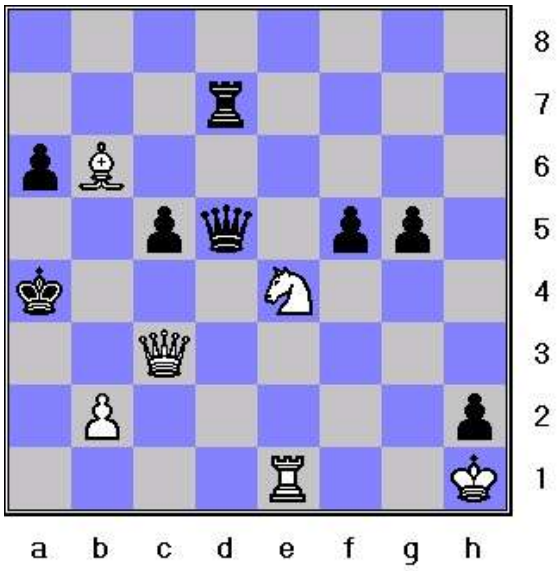


Abb. 5.29: 49. Zug, Übergang ins Endspiel (Nemesis 1.0 – TSCP 1.81)

Nemesis konnte den Gegner durch verschiedene mögliche Angriffskombinationen weiter attackieren: Die Dame steht jetzt sehr gefährlich und der Läufer konnte nach b6 einen gegnerischen Bauern erobern. Er befindet sich, wie der Springer, in einer sehr guten Angriffsposition. Da Weiß mit dem Turm über das Feld a1 ebenfalls angreifen kann, sieht die Situation für TSCP schlecht aus. Der einzige Vorteil, den Schwarz hatte, war der h-Bauer, der aber vom weißen König aufgehalten werden konnte.

50. Bxc5 Rb7 51. Ra1+ Kb5 52. Ra5+ Kc6 53. Ba7+ Qc4 54. Qxc4+ Kd7

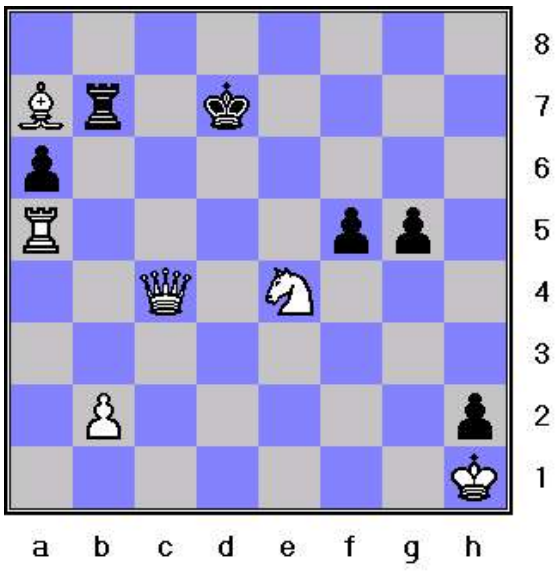


Abb. 5.30: 54. Zug, Endspiel (Nemesis 1.0 – TSCP 1.81)

Weiß hat nun eine drückende Übermacht, die eigenen Figuren sind gut postiert und haben den gegnerischen König eingekreist. Schwarz mußte bereits Material opfern, um einem Schachmatt zu entgehen, der Verlust der Partie ist nicht mehr aufzuhalten.

55. Qf7+ Kd8 56. Rd5+ Kc8 57. Qxf5+ Kc7 58. Qd7# 1 - 0

Nemesis greift weiter mit der Dame und dem Turm an, und TSCP kann im 58. Zug Schachmatt gesetzt werden.

Eine Auswertung dieses Spiels folgt im nächsten Abschnitt. Das zweite Spiel, welches ebenfalls im nächsten Abschnitt ausgewertet wird, ist eine Partie zwischen Nemesis 1.0 und GNUChess 4. Nemesis hat hier mit den schwarzen Figuren gespielt.

Weiß: GNUChess 4 – Schwarz: Nemesis 1.0 (Spiel 4)

1. d4 Nf6 2. c4 e6 3. Nf3 b6 4. a3 Bb7 5. Nc3 d5

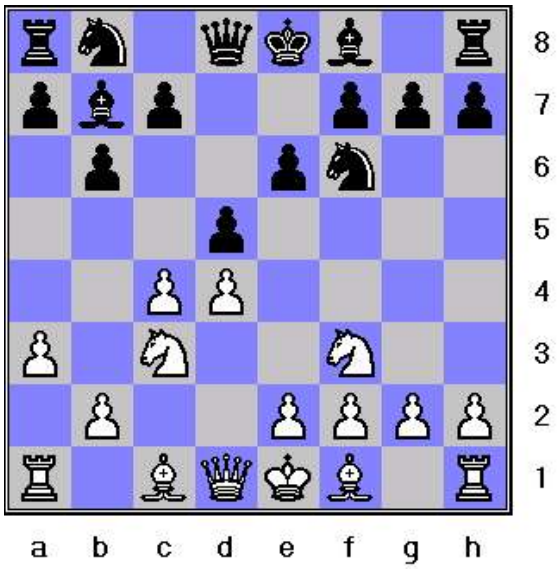


Abb. 5.31: 5. Zug, Eröffnung (GNUChess 4 – Nemesis 1.0)

Die hier gespielten Züge sind als Variante der Nimzo-Indischen Eröffnung bekannt [Chess Archeology], und wurden von beiden Programmen aus der Bibliothek gespielt. Weiß hat eine starke Bauernentwicklung im Zentrum, die Springer üben zusätzliche Kontrolle aus. Der Läufer f1 sollte in den nächsten Zügen entwickelt werden, um den Weg für eine Rochade frei zu machen. Schwarz versucht durch den Bauern d5 das Zentrum zu verteidigen, der durch den Bauern e6 und den Läufer b7 geschützt wird. Der Aufbau des Spiels soll durch den Springer f6 unterstützt werden. Der Weg für eine Rochade kann bald vom Läufer f8 frei gemacht werden.

6. cxd5 exd5 7. g3 Bd6 8. Bh3 O-O 9. Nb5 Re8 10. Nxd6 Qxd6

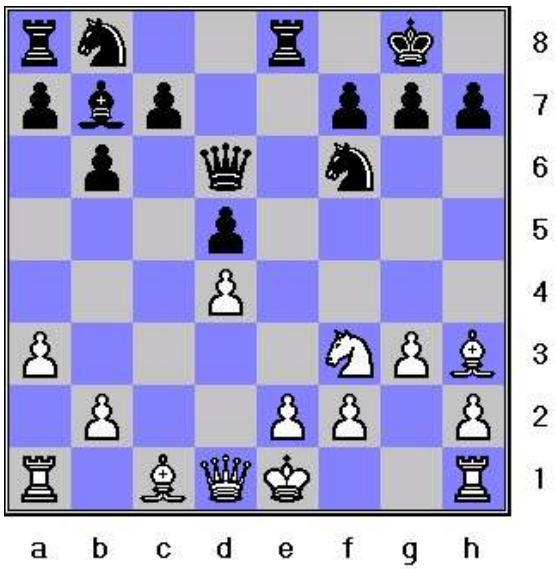


Abb. 5.32: 10. Zug, Übergang ins Mittelspiel (GNUChess 4 – Nemesis 1.0)

Weiß hat seine Stellung nach einem Bauerntausch auf d5 gut entwickelt: Der Läufer h3 hat eine vollständig offene Diagonale, der Läufer c1 ebenfalls. Der Springer f3 greift weiterhin das Zentrum an. Schwarz hat bereits rochiert, um die eigene Stellung zu verstärken, dabei wurde ein Turm nach e8 entwickelt, was eine starke Positionierung auf einer offenen Linie bedeutet. Die Dame d6 kann auf beiden Flügeln angreifen, was durch den Springer f6 unterstützt werden könnte. Der Läufer b7 ist nicht gut entwickelt, da er nur wenig gute Zugmöglichkeiten hat.

11. O-O Ba6 12. Re1 Nbd7 13. Bf4 Qe7

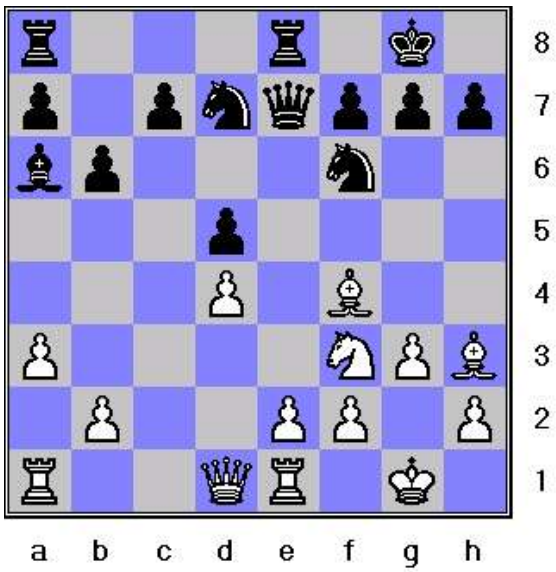


Abb. 5.33: 13. Zug, Übergang ins Mittelspiel (GNUChess 4 – Nemesis 1.0)

GNUChess hat nun ebenfalls rochiert und eine sehr starke Verteidigung aufgebaut. Durch den Zug Läufer nach f4 ist es gelungen, die schwarze Dame aus dem Zentrum zu vertreiben. Weiß hat nun viele Angriffsmöglichkeiten, etwa über die Dame oder den Läufer f4. Schwarz hat seine Stellung leicht entwickeln können, durch Springer b8d7 konnte die eigene Verteidigung verstärkt werden. Läufer nach a6 greift die gegnerische Stellung an.

14. Qa4 Bc4 15. Qc6 Nf8 16. Ne5 Nh5 17. Bd2 Qf6 18. Qxc7 Ne6

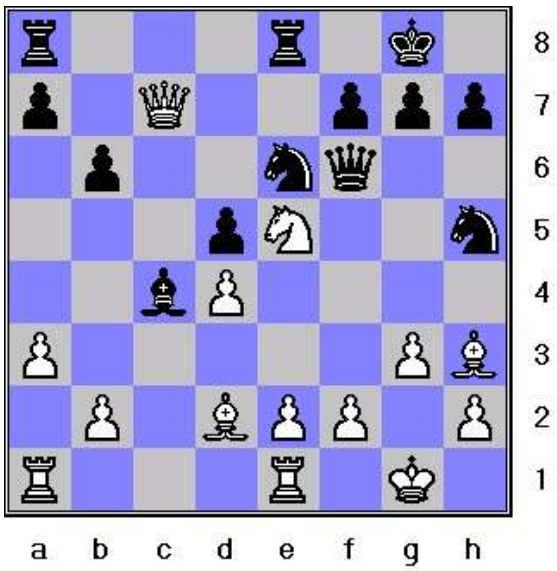


Abb. 5.34: 18. Zug, Verlauf des Mittelspiels (GNUChess 4 – Nemesis 1.0)

Weiß hat über Dame nach a4 einen Angriff gestartet und den Bauern c7 erobern können. Der weiße Springer steht sehr gut auf e5, da er durch einen Bauern geschützt wird. Die beiden Läufer sind ebenfalls sehr gut postiert, da sie nach wie vor Druck auf die Stellung von Schwarz ausüben. Nemesis hat versucht den weißen Angriff mit Läufer nach c4 zu kontern, hat jedoch nicht genügend gut postierte Figuren, um den Angriff fortzusetzen. Der Turm e8 wird durch einen Springer blockiert, was einen Nachteil bedeutet. Insgesamt ist die Stellung von Schwarz nicht so ausgewogen, es gibt weiter wenig Optionen, den Gegner unter Druck zu setzen.

19. Bxe6 Qxe6 20. Nxc4 dxc4 21. Rac1 Rec8 22. Qb7 Nf6 23. e4 Rcb8 24. Qc7 Nxe4 25. Qf4 f5

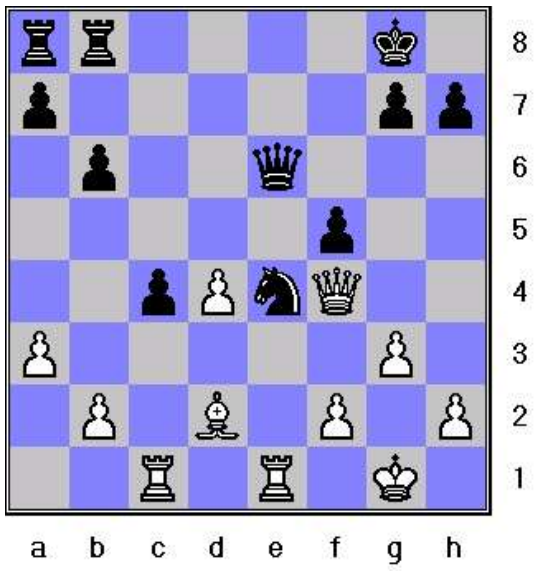


Abb. 5.35: 25. Zug, Verlauf des Mittelspiels (GNUChess 4 – Nemesis 1.0)

Um die Dame von der 7. Reihe zu vertreiben, musste Nemesis beide Türme benutzen, die nun in der Ecke blockiert sind. Weiß hat seine Türme entwickeln können, die beide schwarze Figuren

attackieren. Die gut postierten Läufer wurden getauscht. Schwarz hat kaum Möglichkeiten, die Stellung für sich zu verbessern. Allein der gedeckte Springer auf e4 bringt etwas Entlastung, die eigene Königsdeckung musste jedoch geöffnet werden, um ihn abzusichern.

26. f3 Re8 27. fxe4 fxe4 28. Re3 Qd5 29. Rce1 Qxd4 30. Bc3 Qc5

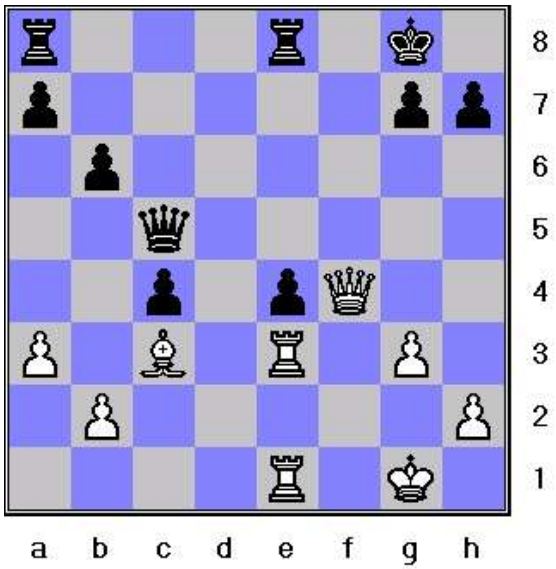


Abb. 5.36: 30. Zug, Verlauf des Mittelspiels (GNUChess 4 – Nemesis 1.0)

Weiß konnte den schwarzen Springer auf Kosten von zwei Bauern erobern. Die beiden Türme sind noch stärker postiert als zuvor, da sie zusammen auf einer Linie angreifen. Der Läufer auf c3 ist ebenfalls sehr stark, da er Angriffe auf die schwarze Königsdeckung unterstützen kann. Schwarz hat sich nicht vom gegnerischen Druck befreien können, die Dame c5 fesselt den Turm e3, hat jedoch keine weiteren Entwicklungsmöglichkeiten. Die Königsdeckung ist durch den Verlust des f-Bauern schwächer geworden; die Türme bieten zwar einen gewissen Schutz, können aber so nicht mehr für den Angriff genutzt werden.

31. Kh1 Rad8 32. Rxe4 Qc6 33. Kg1 Qc5+ 34. Bd4 Qxd4+ 35. Rxd4 Rxe1+ 36. Kf2 Rde8 37. Rd7 R1e2+ 38. Kf3 R2e7 39. Qxc4+ Kf8 40. Rd2 g6

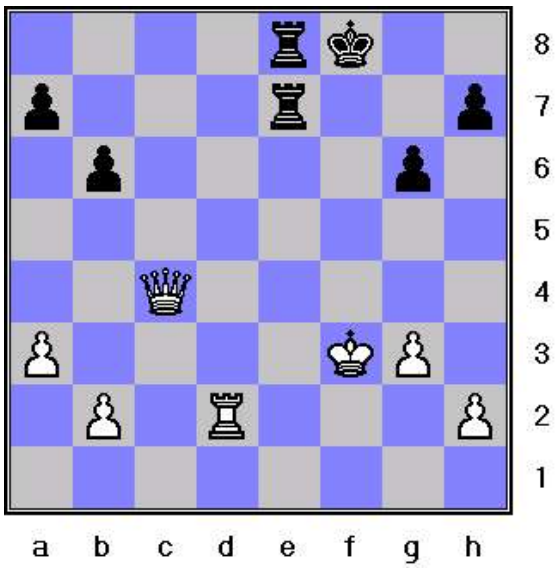


Abb. 5.37: 40. Zug, Übergang ins Endspiel (GNUChess 4 – Nemesis 1.0)

GNUChess hat aufgrund des starken Drucks auf den schwarzen König seinen Läufer gegen die schwarze Dame tauschen können, was einen deutlichen Materialgewinn bedeutet. Diese Vereinfachung verzögert zwar den Spielablauf, macht es aber gleichzeitig für Schwarz äußerst schwierig, noch einmal in die Offensive zu kommen: Nemesis hat beide Türme nah am eigenen König postieren müssen, um diesem Schutz zu geben. Optionen für die Entwicklung der Stellung gibt es dadurch nicht mehr.

41. Kg2 Kg7 42. Rf2 Re6 43. Qf4 Kh8 44. b4 R6e7 45. Qd4+ Kg8 46. Qf6 Rd7 47. h4 Rc7 48. Qf4 Rce7 49. Qc4+ Kg7 50. Qd4+ Kg8 51. Qf4 h5

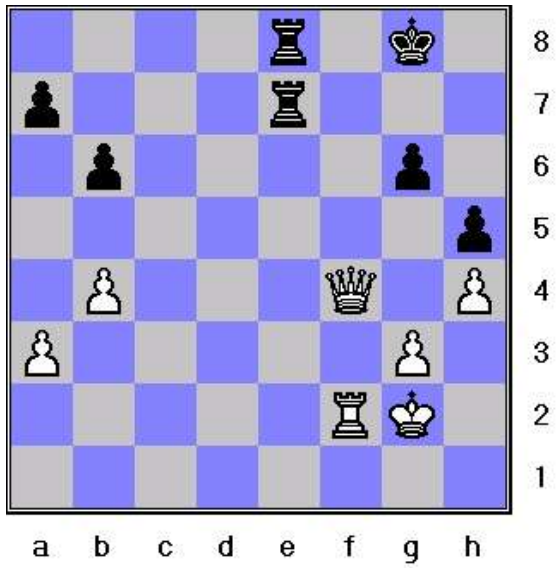


Abb. 5.38: 51. Zug, Endspiel (GNUChess 4 – Nemesis 1.0)

Weiß hat sich nun in die notwendige Position gebracht, um die finale Attacke auszuführen. Nemesis hat diesen Ablauf über 10 Züge verzögern können, hat jedoch keine Möglichkeiten mehr, einen wirkungsvollen Gegenangriff auszuführen.

52. Qh6 Re6 53. b5 Rd6 54. Rc2 Re7 55. Rc8+ Kf7 56. Rf8+ Ke6 57. Qxg6+ Kd7 58. Qxh5 Rd4 59. Qf5+ Re6 60. Kh3 Rd1 61. Rf7+ Kd6 62. Qf4+ Re5 63. Qf6+ Kd5 64. Qc6+ Kd4 65. Rd7+ Ke3 66. Rxd1 Re7

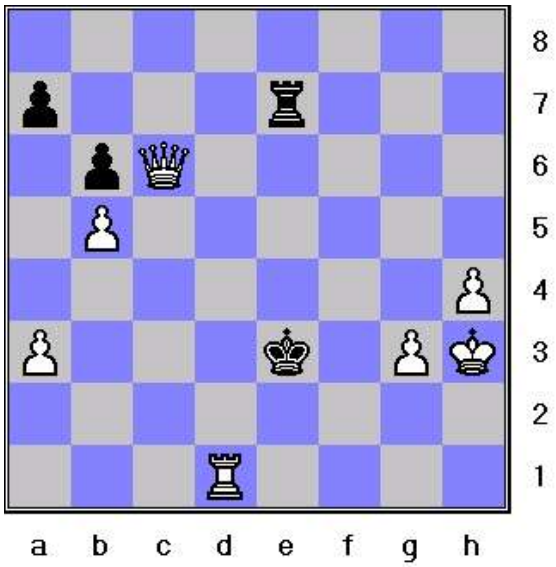


Abb. 5.39: 66. Zug, Endspiel (GNUChess 4 – Nemesis 1.0)

GNUChess hat weitere 14 Züge gebraucht, um einen der Türme zu schlagen und den schwarzen König zu entblößen. Für Nemesis ist keine Verteidigung mehr möglich.

67. Qc2 Rd7 68. Re1+ Kd4 69. Re4+ Kd5 70. Qc6# 1 – 0

Die kombinierten Angriffe der Dame und des Turms führen im 70. Zug zum Schachmatt.

5.4.3. Bewertung der Testspiele

Ich werde nun am Beispiel der beiden Testspiele erläutern, welchen Einfluß sowohl eine hohe Suchgeschwindigkeit, als auch eine gute Bewertungsfunktion auf die Spielstärke der Programme haben. An diesen Ergebnissen werden die spezifischen **Unterschiede** zwischen Nemesis 1.0 einerseits und TSCP 1.81 und GNUChess 4 andererseits deutlich: Nemesis und TSCP erreichen trotz unterschiedlicher Plattformen die gleichen Suchtiefen, die deutlichen Resultate zugunsten von Nemesis sind also auf die Auswirkungen der Bewertungsfunktion zurückzuführen, die zu einem überlegenen positionellen Spiel verhilft. GNUChess und Nemesis wiederum zeigen ein ausgeglichenes positionelles Spiel, GNUChees erreicht jedoch deutlich höhere Suchtiefen, und konnte deswegen alle Spiele für sich entscheiden. Die Unterschiede der verschiedenen Schachprogramme lassen sich durch alle Partien bestätigen, da alle die genannten Eigenschaften gezeigt haben (s. Anhang C „Testspiele“ sowie beiliegende CD).

Die erste Partie zeigt den großen Einfluß der **Bewertungsfunktion**: Nemesis bewertet positionelle Faktoren stark, und zeigt dadurch bereits in der Eröffnung (die ohne Bibliothek gespielt wurde), sowie im Übergang zum Mittelspiel dieser Partie eine deutlich bessere Entwicklung als TSCP (Abb. 5.24). Nemesis hat bereits hier durch die Faktoren King Tropism der Dame und King Pressure der Dame und des Läufers f4 erkannt, daß ein gefährlicher Angriff auf die Stellung des Gegners möglich ist. Nemesis bewertet bereits hier mit +164 Punkten. TSCP verhält sich zu passiv, und erkennt die drohende Gefahr eines Angriffs auf seine Königsstellung nicht. Nach Abtausch der Türme hat sich Nemesis noch besser postiert (Abb. 5.26), da viele Figuren Angriffe auf die gegnerische Stellung ausführen können, also entweder in King Tropism oder King Pressure belohnt werden. Der Verlust der Bauerndeckung wird zwar negativ bewertet, da Schwarz jedoch keine eigene King Pressure aufbauen kann, wird der Angriff mit einer Bewertung von +224 Punkten fortgesetzt. Bereits bevor die Dame über a8 Schach gesetzt hat, hat Nemesis in der Suche erkannt, daß Material gewonnen werden kann. Dies ist vor allem durch die Verwendung von Search Extensions möglich, die bei solchen Angriffen vielfach ausgelöst werden; Nemesis gelingt es dann auch tatsächlich, den gegnerischen Läufer zu schlagen (Abb. 5.28). Die Fortführung des Angriffes zeigt sehr gute Suchergebnisse (+581 Punkte), die durch die Entwicklung der Stellung bestätigt werden (Abb. 5.29). Der folgende Mattangriff kann ohne Gefahr ausgeführt werden, da Nemesis bereits eine deutliche Überlegenheit in Material und Positionierung erreicht hat.

Die Partie zwischen GNUChess 4 und Nemesis 1.0, die ich dargestellt habe, zeigt einen deutlich langwierigeren Verlauf. Beide Programme erreichen eine gute Bewertung positioneller Faktoren, da sich diese nur sehr langsam zu Gunsten einer Seite verschieben: Erst ab dem 18. Zug ist eindeutig erkennbar, daß Weiß etwas besser postiert ist als Schwarz (Abb. 5.34), was von Nemesis mit einem Suchergebnis von -56 erkannt wird. GNUChess gelingt es nun, diesen Vorteil weiter auszubauen, was vor allem möglich ist, da es deutlich größere **Suchtiefen** erreicht: Der Fortgang der Partie kann dadurch wesentlich genauer bestimmt werden, was die Möglichkeit ergibt, den Druck auf den Gegner zu erhöhen. Nemesis gerät immer mehr in die Defensive (Abb. 5.35) und erkennt dies durch ein Suchergebnis von -184. Aufgrund der geringeren Suchtiefe gibt es aber kaum eine Chance, eigene Angriffe einzuleiten, da alle Möglichkeiten, die hierfür erkannt werden können, bereits vorher von GNUChess vereitelt werden. Schwarz versucht einen letzten Gegenangriff, der aber durch den Verlust des Springers zu einem Ende kommt (Abb. 5.36). Der mögliche Materialgewinn konnte von GNUChess zwei Züge früher entdeckt werden, was zu einem noch größeren Ungleichgewicht der Stellung geführt hat. Auf dieser Basis gelingt es Weiß den Druck auf die schwarze Stellung zu erhöhen, bis es in 5.37 eine deutlich Materialüberlegenheit erreicht hat. Nemesis bewertet hier bereits mit -456 Punkten und kann nichts anderes mehr tun, als den Verlust weiteren Materials möglichst lange zu verzögern, was immerhin 15 Züge lang gelingt. Nachdem jedoch ein Turm verloren wurde (Abb. 5.39), ist der Verlust der Partie nicht mehr abzuwenden.

Neben der dargelegten Analyse der Ursachen unterschiedlicher Spielweisen, kann man aus den Resultaten der Testspiele auch eine quantitative Bewertung entnehmen: Wenn man die **Spielstärke** von Nemesis anhand der Resultate der Testspiele bestimmt, so kann man einen ungefähren ELO-Wert von 1900-2000 Punkten annehmen. Dieses Ergebnis ergibt sich aus der Tatsache, daß Nemesis gegen TSCP kein Spiel verloren hat, und drei von vier Spielen gewinnen konnte. Man kann also davon ausgehen, daß Nemesis eine höhere Spielstärke besitzt. Da TSCP mit 1867 ELO-Punkten bewertet wird, ergibt sich hieraus, daß der ELO-Wert von Nemesis oberhalb dieses Wertes liegen muß[35]. Der ELO-Wert von GNUChess 4 liegt bei 2075 Punkten. Da Nemesis gegen diesen Gegner alle Partien verloren hat, liegt Nemesis' ELO-Wert darunter. Aus diesen Vergleichen ergibt sich meine Schätzung, daß der ELO-Wert von Nemesis 1.0 bei 1900-2000 Punkten liegt. Dieser Wert würde nach der offiziellen Klassifizierung der ELO-Werte [Wikipedia Elo] dem oberen Bereich der Spielstärke „Amateur, Klasse A, sehr guter Vereinsspieler“ entsprechen. Die Anforderung nach einer hohen Spielstärke der zu entwickelnden Schachanwendung, die ich in der Einleitung auf die Spielstärke eines guten Vereinsspielers festgelegt hatte, konnte also voll erfüllt werden.

35 Man kann diesen Wert nach statistischen Verfahren errechnen, es sind jedoch deutlich mehr Partien notwendig, um ein realistisches Ergebnis zu erhalten [Wikipedia Elo]

6. Zusammenfassung und Fazit

6.1. Erfüllung der Anforderungen

In diesem Abschnitt werde ich darstellen, ob und wie die in Kapitel 4.1 formulierten Anforderungen an die Zielumgebung und die entwickelte Schachanwendung erfüllt werden konnten. Hierbei ist zu unterscheiden, daß die Erfüllung der Anforderungen an die Zielumgebung vor allem durch die Auswahl geeigneter Werkzeuge gefördert werden konnte, da die konkrete Zielplattform vom Verfasser nicht selber ausgewählt wurde. Die Anforderungen an die Schachanwendung hingegen mußten durch die Implementation von Nemesis 1.0 selbständig erfüllt werden.

Ich werde die verschiedenen Anforderungen hier erneut darstellen, um die Ergebnisse der Arbeit in Bezug zu ihnen stellen zu können:

1. Anforderungen an die Zielumgebung

1.1. Die Zielumgebung soll eine benutzerfreundliche Entwicklung des Schachprogramms ermöglichen: Diese Anforderung konnte insgesamt erfüllt werden. Die Entwicklungsumgebung Microsoft eMbedded Visual C++ 3.0 bietet komfortable Möglichkeiten zur Entwicklung einer Anwendung für Pocket PCs und ist kostenfrei. Der HP iPAQ H5500 ist Multitasking-fähig, einfach zu bedienen und besitzt ein ausgereiftes Betriebssystem.

1.2. Die Zielumgebung soll gute Debugging-Möglichkeiten besitzen: Diese Anforderung konnte kaum realisiert werden. eMbedded Visual C++ 3.0 bietet hierzu zwar gewisse Möglichkeiten, es wird jedoch kein Exception Handling unterstützt, was einen ernsthaften Nachteil darstellt. Die Entwicklung auf dem Emulator ist ebenfalls problematisch, da sich sein Verhalten von der Zielplattform deutlich unterscheidet. Der verwendete PDA ist in diesem Sinne auch nicht gut geeignet, da durch die geringe Größe des Displays nur sehr kleine Datenmengen ausgegeben werden können.

1.3. Die Zielumgebung soll ein Schachspiel benutzerfreundlich darstellen können: Diese Anforderung konnte trotz des kleinen Displays des HP iPAQ H5500 mit Hilfe von CEBoard 2.1 zufriedenstellend realisiert werden.

1.4. Die Zielplattform soll eine möglichst große Speicherumgebung zur Verfügung stellen: Diese Anforderung hat die größten Probleme gezeigt. Der HP iPAQ H5500 stellt nur 64 MB RAM zur Programmausführung bereit, was im Rahmen einer Schachanwendung zu wenig ist, um einen optimalen Ablauf der Verfahren zu ermöglichen. Interessant wäre ein Vergleich mit einem gleichen Gerät mit größerer Speicherumgebung gewesen, dies konnte im Rahmen dieser Arbeit jedoch nicht realisiert werden.

1.5. Die Zielplattform soll einen möglichst hohen Systemtakt besitzen, d.h. hohe Rechenleistungen erbringen können: Diese Anforderung wird vom HP iPAQ H5500 erfüllt. Natürlich erreichen stationäre Systeme höhere Rechenleistungen, im Rahmen der in dieser Arbeit betrachteten Plattformen stellt der HP iPAQ H5500 jedoch ein leistungsfähiges Gerät dar. Gewisse Einschränkungen der Rechenleistung in diesem ressourcenarmen System werden jedoch deutlich, wenn man den Vergleich der Testergebnisse auf dem PC mit und ohne Emulator betrachtet.

2. Anforderungen an die Schachanwendung

2.1. Die Schachanwendung soll in der Lage sein, eine Schachpartie regelkonform zu bestreiten, d.h. sie darf keine illegalen Züge ausspielen: Diese Anforderung konnte durch die Beachtung der in 2.1.6. dargestellten Ansätze berücksichtigt werden, ohne zu große Performanzverluste zu erzeugen.

2.2. Die Schachanwendung soll ein stabiles Suchverhalten zeigen, das in allen (vergleichbaren) Tests die gleiche Performanz erreicht: Diese Anforderung konnte zufriedenstellend erfüllt werden, wie ein Vergleich der erreichten Suchtiefen in den Testspielen gezeigt hat.

2.3. Die Schachanwendung soll ein korrektes Suchverhalten zeigen, also keine falschen Suchergebnisse erzeugen: Diese Anforderung konnte voll erfüllt werden. Die Ergebnisse der Teststellungen haben 94% erfolgreiche Lösungen auf dem Emulator gezeigt, was das Vorhandensein falscher Suchergebnisse sehr unwahrscheinlich erscheinen läßt.

2.4. Die Schachanwendung soll in der MinMax-Suche möglichst hohe Suchtiefen erreichen: Diese Anforderung konnte mutmaßlich erfüllt werden. Im Vergleich zu TSCP 1.81 konnten gute Werte erzielt werden, da Nemesis trotz deutlich langsamerer Plattform die gleichen Suchtiefen erreicht hat. GNUChess 4 erreicht auf einer schnelleren Plattform als Nemesis deutlich höhere Suchtiefen, woraus kaum zu entnehmen ist, welche weiteren Verbesserungen für Nemesis möglich wären.

2.5. Die Schachanwendung soll ein gutes positionelles Spiel zeigen: Diese Anforderung kann anhand der Ergebnisse der Testspiele als vollständig erfüllt betrachtet werden. Nemesis 1.0 hat hier gezeigt, daß es ein deutlich stärkeres positionelles Spiel besitzt als TSCP und im Vergleich zu GNUChess ebenfalls akzeptable Spielverläufe erreicht[36]. Die Ergebnisse von Teststellungen haben ebenfalls gezeigt, daß Nemesis viele Lösungen frühzeitig erkennen konnte, da sie über die Bewertungsfunktion antizipiert wurden.

2.6. Die Schachanwendung soll eine möglichst hohe Spielstärke besitzen: Diese Anforderung konnte erfüllt werden. Nemesis hat 84% der Teststellungen lösen können, was eine hohe Qualität der Suche und der Bewertungsfunktion zeigt. In Testspielen konnte gezeigt werden, daß Nemesis gegen einen Gegner der Spielstärke „sehr guter Vereinsspieler“ gewinnt, also mindestens die gleiche Spielstärke besitzt. Gegen GNUChess waren keine Siege möglich, weil dieses Programm aufgrund der Spielstärke eines „Schachexperten“ einen zu starken Gegner darstellt.

36 Es wurden zwar alle Spiele verloren, jedoch nicht aufgrund mangelnden positionellen Verständnisses

6.2. Ergebnisse der Arbeit

Die Zielsetzung der Arbeit war, zu evaluieren, welche Möglichkeiten zur Implementation einer rechenintensiven Anwendung in einem aktuellen ressourcenarmen System bestehen und welche Leistung die Anwendung dort erbringen kann. Um dies zu bewerten, wurde ein Schachprogramm für einen Hewlett Packard iPAQ H5500 PDA entwickelt. Zur Implementation der Anwendung wurden die theoretischen Grundlagen sowie die verwendeten Verfahren erläutert und es wurde vorgestellt, wie diese Verfahren in die Praxis umgesetzt wurden. In der Bewertung unterschiedlicher Optimierungsmöglichkeiten wurde dargestellt, daß es wichtig ist, die richtigen Verfahren auszuwählen und zu kombinieren, da ein optimaler Ablauf der MinMax-Suche nicht von der Verwendung eines bestimmten Verfahrens abhängt. Ich habe verdeutlicht, daß diese Auswahl durch die technischen Möglichkeiten der Zielplattform maßgeblich beeinflußt wird, da diese die mögliche Effizienz der Verfahren bestimmen. Darauf aufbauend habe ich sowohl theoretisch als auch praktisch gezeigt, welche Verfahren für eine Schachanwendung in ressourcenarmen Umgebungen besonders wichtig sind, und wie ich zu dieser Einschätzung gekommen bin.

In Kapitel 5 habe ich die Fähigkeiten des Schachprogramms Nemesis 1.0 getestet und bin dabei zu dem Ergebnis gekommen, daß es möglich ist, auf einem ressourcenarmen System eine Schachanwendung zu realisieren, die gute Resultate erzielt: In Tests konnte gezeigt werden, daß Nemesis in der Lage ist, 85% einer Menge gegebener Teststellungen zu lösen. Das nicht mehr Lösungen gefunden wurden, liegt zum Teil an den begrenzten Ressourcen der Plattform, was durch Testergebnisse auf dem Emulator bewiesen werden konnte. Der Vergleich zu den Testresultaten der PC-Version von Nemesis hat klar gemacht, daß die Zielplattform eigenen Overhead verursacht, der nicht durch die offensichtlichen technischen Beschränkungen erklärt werden kann.
Die Testspiele gegen TSCP 1.81 und GNUChess 4 haben gezeigt, daß Nemesis auf dem iPAQ H5500 aufgrund einer leistungsfähigen Suche und einer guten Bewertungsfunktion eine Spielstärke besitzt, die ausreichend ist, um einen Gegner auf dem Niveau eines erfahrenen Vereinsspielers zu schlagen. Professionellen Spielern sollte Nemesis immer noch eine Herausforderung bieten können, obwohl es auf höchstem Niveau sicher nicht mithalten kann. Insgesamt sind die Resultate von Nemesis 1.0 sehr zufriedenstellend, da die anvisierte Spielstärke tatsächlich erreicht werden konnte. Dies kann durch einen geschätzten ELO-Wert von 1900-2000 Punkten belegt werden; die nächste Stufe des ELO-Wertungssystems entspricht bereits einem „Meisteranwärter“ bzw. „Schach-Experten“.

Die Zielplattform hat sich als für eine Schachanwendung gut geeignet herausgestellt, wie die Ergebnisse der Evaluation belegen. Die größten Einschränkungen waren durch eine geringe Speicherumgebung sowie die Größe des Displays gegeben. Den Mangel an Speicher habe ich versucht durch eine geschickte Auswahl der verwendeten Verfahren zu kompensieren und konnte zeigen, daß der Hewlett Packard iPAQ H5500 PDA immer noch genügend Ressourcen bietet, um als Plattform für ein spielstarkes Schachprogramm zu fungieren. Hierdurch ist deutlich geworden, daß bei entsprechender Rücksichtnahme auf die technischen Gegebenheiten, auch auf einem ressourcenarmen System eine rechenintensive Anwendung erfolgreich implementiert werden kann. Zwar ist ein gewisser Overhead auf der Zielplattform zu beobachten, wie in 5.3.7. gezeigt wurde, der aber bei einer sorgfältigen Implementation der Anwendung nicht entscheidend ausfällt. Die geringe Größe des Displays ist ein Faktor, der durch die Implementation der Anwendung nicht zu beheben war, glücklicherweise konnte mit CEBoard 2.1 jedoch eine Möglichkeit gefunden werden, das bestmögliche Ergebnis zu erreichen.

Wie bereits in Abschnitt 5.3.6. erwähnt wurde, ist die Entwicklung und Optimierung eines Schachprogramms für ein ressourcenarmes System ein gutes Beispiel für das Pareto-Prinzip: Ein Großteil der erreichten Performanz konnte durch die Implementation des PVS/NegaScout-

Algorithmus mit MVV/LVA- / Killer – Vorsortierung und Transposition Table realisiert werden, was für den Entwickler eine überschaubare Herausforderung darstellte. Die restlichen Optimierungen, die noch erreicht werden konnten, entfallen auf alle weiteren Verfahren wie Verified Nullmove Pruning, Iterative Deepening, Aspiration Search, Search Extensions, Futility Pruning und Lazy Eval. Hier gibt es eine große Menge weiterer Ansätze, deren Implementation ein sehr genaues Verständnis der Anwendung erfordert. Man kann also zusammenfassend sagen, daß die Implementation eines Schachprogramms an sich mit wenig Aufwand geleistet werden konnte. Erst die Optimierung des Suchverhaltens, die aufgrund der gegebenen Einschränkungen der Zielplattform notwendig war, hat zu einem großen Aufwand geführt, der für die Implementation weiterer Verfahren geleistet werden musste. Man kann dies als eine inverse Relation zwischen den Ressourcen der Zielplattform und des zur Implementation einer rechenintensiven Anwendung nötigen Aufwands interpretieren.

Die Entwicklung der Anwendung hat sich dadurch auch als eine reizvolle Aufgabe herausgestellt: Es ist dem Entwickler überlassen gewesen, aus der großen Menge bekannter Verfahren eine möglichst gute Auswahl zu treffen. Die Implementation dieser Verfahren ließ ebenfalls Spielräume für eigene Optimierungen offen, so daß das Schachprogramm für ressourcenarme Systeme ein originäres Produkt des Entwicklers darstellt. Entwurf und Umsetzung einer Bewertungsfunktion für Nemesis 1.0 waren unter diesem Gesichtspunkt besonders interessant, da hierfür eigene kreative Ansätze entwickelt werden konnten, deren Resultate maßgeblich sowohl die Spielstärke als auch die Spielweise des entwickelten Schachprogramms beeinflußt haben.

6.3. Ausblick

Die Verbreitung mobiler Plattformen schreitet rasant fort: 2004 wurden weltweit 23,8 Mio. Geräte der PDA - bzw. Smartphone – Klasse verkauft, für 2006 werden Verkäufe von insgesamt 38,3 Mio. Geräten prognostiziert [eTForecasts]. Im ersten Halbjahr 2005 wurden außerdem bereits 25,6 Mio. mobile Geräte zur digitalen Musikwiedergabe verkauft [Canalys]. Diese Zahlen zeigen deutlich, daß die Anzahl mobiler digitaler Endgeräte im Konsumentenmarkt stark zunimmt. Wenn man die Entwicklung der mobilen Telekommunikation als Vergleich nimmt, so kann man davon ausgehen, daß in näherer Zukunft mobile Computer eine ähnliche Verbreitung erreichen werden wie Handys heutzutage. Es könnte auch die angedeutete Entwicklung eintreten, daß diese verschiedenen Geräteklassen aufgrund des in 3.3. gezeigten technischen Fortschritts zu einer verschmelzen.
An den Ergebnissen aus Kapitel 5 ist deutlich geworden, daß Nemesis eine hohe Spielstärke erreicht, weil die hierfür notwendige Leistungsfähigkeit bereits durch eine aktuelle Plattform aus dem PDA-Segment gegeben ist. Da ein ressourcenarmes Gerät also in der Lage ist, die Anforderungen einer Schachanwendung zufriedenstellend zu erfüllen, wird deutlich, daß Geräte dieser Klasse bereits Aufgaben übernehmen können, die in der Vergangenheit nur von Großrechenanlagen geleistet werden konnten. Wenn man nun eine diesem Vergleich entsprechende Steigerung der Leistungsfähigkeit mobiler Geräte für die Zukunft prognostiziert, so scheint es, als wenn stationäre Computer tatsächlich überflüssig werden könnten. Sie wären durch mobile Systeme, die natürlich weitere Vorzüge bieten, ersetzbar. Diese hochintegrierten Plattformen könnten überall genutzt werden, um so verschiedene Anforderungen wie Datenverwaltung, Telekommunikation, Musikwiedergabe, digitale Photographie und eben auch rechenintensive Anwendungen auf hohem Niveau miteinander zu verbinden. Eine solche Entwicklung wäre für das Gebiet des Ubiquitous Computing besonders interessant: Aktuell ist eine rechenintensive Anwendung auf einem ressourcenarmen System möglich, wenn man nun die Idee des Ubiquitous Computing weiter verfolgt, so scheint es für die Zukunft denkbar, eine solche Anwendung in einer Menge vernetzter Kleinstcomputer zu realisieren und eine hohe Leistungsfähigkeit zu erreichen.

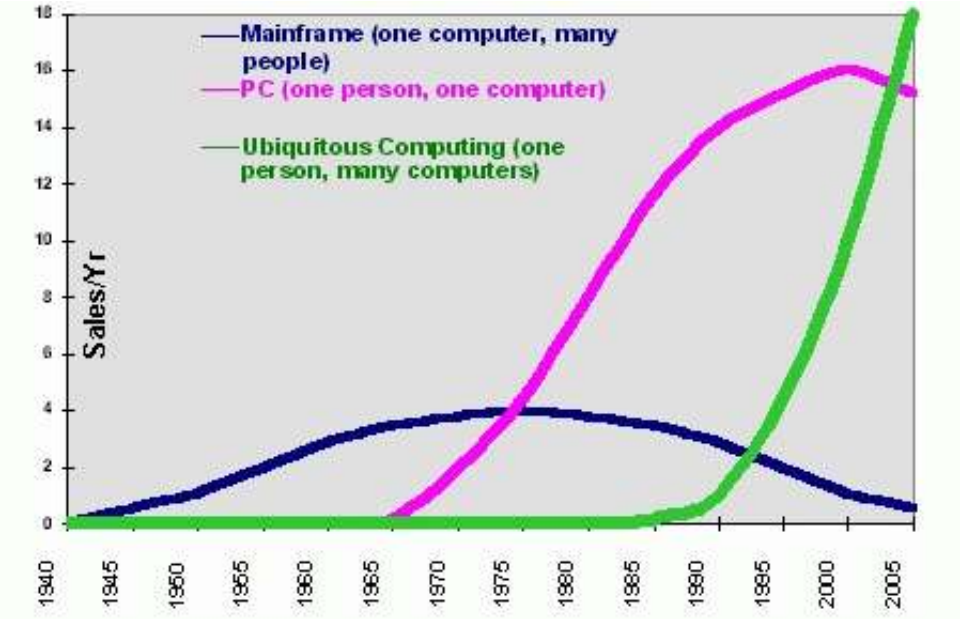


Abb. 6.1: Verkäufe von unterschiedlichen Computer-Plattformen [Xerox PARC]

Anhang A: Quellennachweis

Quellen die mit 'CD' markiert sind, sind unter dem angegebenen Namen auf der beiliegenden CD verfügbar (s. Anhang B). Alle URLs sind am 10.10.2005 das letzte Mal auf Erreichbarkeit geprüft worden.

[Ars Electronica] Ars Electronica Archiv: *Die Teilnehmer an der 3. Computerschach-Weltmeisterschaft*, URL http://www.aec.at/de/archives/festival_archive/festival_catalogs/festival_artikel.asp?iProjectID=9497

[Beal 89] Beal, D.: *Experiments with the Nullmove*, in: *Advances in Computer Chess 5*, Beal, D. (Hrsg.), Elsevier, 1989

[Brudno 63] Brudno, A.L.: *Bounds and Valuations for Abridging the Search of Estimates*, in: *Problems of Cybernetics 10*, 1963

[Canalys] Canalys Research releases: *Portable music player market expected to double by end of 2005*, 2005, URL http://www.canalys.com/pr/2005/r2005091.htm

[Chess Archeology] Chess Archeology: *Chess Openings classified by ECO code*, 2001, URL http://www.chessarch.com/library/0000_eco/e20_e59.shtml

[Chess Captor] The Chess Captor Homepage: *Downloading Chess Captor*, 2004, URL http://www.chesscaptor.com/download.php

[Chess War] Comité Loire Echecs: *ChessWar VII C 40m/20'*, 2004, URL http://loirechecs.chez.tiscali.fr/chesswar/Chesswar007/Chesswar007CLs.htm

[Chip.de] Chip.de: *Test: Dell Axim X30 - Schnellster PDA der Welt?*, 2005, URL http://www.chip.de/artikel/c1_artikel_12833607.html

[Computerwoche] Computerwoche, Heftarchiv: *Die Schachprogramme lernen dazu*, URL http://www3.computerwoche.de/heftarchiv/1978/19780303/a116182.html

[Cray-Cyber Team] Cray-Cyber Team: *Pictures of the Cray-Cyber Computer Rooms*, 2003, URL http://www.cray-cyber.org/pictures/general1.php

[Dubois] Dubois, Claude: *Le Système du Suisse Saison n°3*, 2004, URL http://perso.wanadoo.fr/lefouduroi/tournois/sui/sui.htm

[Eppstein] Eppstein, David: *ICS 180A, Spring 1997: Strategy and board game programming – Lecture notes*, 1999, URL http://www.ics.uci.edu/~eppstein/180a/970401.html, CD (Eppstein.pdf)

[eTForecasts] eTForecasts Research and Consulting: *Smartphones Have Started to Impact PDA Sales*, 2005, URL http://www.etforecasts.com/pr/pr0603.htm

[ETH Ubicomp] Eidgenössische Technische Hochschule Zürich: *What is Ubiquitous Computing?*, 2005, URL http://www.vs.inf.ethz.ch/publ/ubicomp.html

[Freescale] Freescale Semiconductor: *Freescale Smartphones*, 2005, URL http://www.freescale.com/webapp/sps/site/application.jsp?nodeId=02XPgQ425559md

[Futuremark] Futuremark Corporation: *Smartphone Benchmark*, 2004, URL http://www.futuremark.com/companyinfo/SPMark04_Whitepaper.pdf, CD (Futuremark.pdf)

[Ginsberg 93] Ginsberg, Matthew L.: *Essentials of Artificial Intelligence*, Morgan Kaufmann Publishers, 1993

[GNU] GNU Project: *GNUChess Homepage*, 2002, URL http://www.gnu.org/software/chess/chess.html

[Goetsch; Campbell 90] Goetsch, G.; Campbell, M.: *Experiments with the Nullmove Heuristic*, in: *Computers, Chess and Cognition*, T.A. Marsland (Hrsg.), Springer-Verlag, 1990

[Hartz 05] Hartz, Johannes: *Anwendung von Nullfenster-Suchverfahren in der Schachprogrammierung*, Studienarbeit, Hochschule für Angewandte Wissenschaften Hamburg, 2005, CD (Hartz_05.pdf)

[Hewlett Packard] Hewlett Packard Homepage: *HP iPAQ H5500 Pocket PC series*, 2005, URL http://h200003.www2.hp.com/bizsupport/TechSupport/DocumentIndex.jsp?locale=de_DE&contentType=SupportManual&prodTypeId=215348&prodSeriesId=322916&docIndexId=3124&manualLang=en

[HTC] HTC Corporation: *Products - Smartphones*, 2005, URL http://www.htc.com.tw/company/products.html

[Intel Techdocs] Intel Corporation: *Intel PXA255 Processor Technical Documents*, URL http://www.intel.com/design/pca/products/pxa255/techdocs.htm

[Intel XScale] Intel Corporation: *Intel XScale Technology*, URL http://www.intel.com/design/intelxscale/

[Kerrigan] Kerrigan, Tom: *Tom Kerrigan's Homepage*, URL http://home.comcast.net/~tckerrigan/

[Knuth; Moore 75] Knuth, D.E.; Moore, R.W.: *An Analysis Of Alpha/Beta Pruning*, Artificial Intelligence, Band 6, S. 293-326, 1975

[Lazar 95] Lazar, Sashi: *Analysis of Transposition Tables and Replacement Schemes*, Department of Computer Science and Electrical Engineering, University of Maryland, 1995, CD (Lazar_95.pdf)

[Levy 76] Levy, David: *Chess and Computers*, S. 54-57, Batsford, London, 1976

[Loiodice] Loiodice, John B.: *Loiodice's Chess Collection*, URL http://loiodice.com/chess/mychessviewer22/cc-winchess.sht

[Marsland 83] Marsland, T.A.: *Relative Efficiency of Alpha-Beta Implementations*, in: *Procs. 8th International Conference on Artificial Intelligence*, Karlsruhe, 1983

[Marsland 91] Marsland, T.A.: *Computer Chess and Search*, Computing Science Department, University of Alberta, 1991, CD (Marsland_91.pdf)

[Marsland; Campbell 82] Marsland, T.A.; Campbell, M.: *Parallel Search Of Strongly Ordered Game Trees*, Computing Surveys 14(4), 1982

[Meyers 77] Bibliographisches Institut, Mannheim (Hrsg.): *Meyers Enzyklopädisches Lexikon*, Neunte Auflage, Bd. 21, S. 786 f., Mannheim/Wien/Zürich, 1977

[Microsoft Mobile] Microsoft Corporation: *What's New in Windows Mobile 2003 Second Edition Software*, 2004, URL http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnppcgen/html/whatsnew2003se.asp

[MSN Encarta] Microsoft Network Encarta: *Deep Blue and Garry Kasparov*, 2005, URL http://encarta.msn.com/media_461573705_761563087_-1_1/Deep_Blue_and_Garry_Kasparov.html

[Neumann v.; Morgenstern 44] Neumann, John von; Morgenstern, Oskar: *Theory of Games and Economic Behavior*, Princeton University Press, 1944

[Nokia] Nokia Europe: *Nokia 9300 Smartphone Features*, 2005, URL http://www.nokia.com/nokia/0,,60765,00.html

[Plaat; Schaeffer; Pijls; de Bruin 94] Plaat, Aske; Schaeffer, Jonathan; Pijls, Wim; de Bruin, Arie: *A New Paradigm forMinimax Search*, University of Alberta technical report 94-18, 1994, CD (Plaat_Schaeffer_Pijls_deBruin_94.pdf)

[Plaat; Schaeffer; Pijls; de Bruin 95] Plaat, Aske; Schaeffer, Jonathan; Pijls, Wim; de Bruin, Arie: *Best-First Fixed-Depth Minimax Algorithms*, University of Alberta technical report, 1995, CD (Plaat_Schaeffer_Pijls_deBruin_95.pdf)

[Rebel] Rebel Free Software: Opening Book Section, URL http://www.rebel.nl/edsoft.htm

[Reinefeld 83] Reinefeld, Alexander: *An Improvement of the Scout Tree-Search Algorithm*, International Computer Chess Association Journal 6(4), 1983, CD (Reinefeld_83.pdf)

[Reinefeld 89] Reinefeld, Alexander: *Spielbaum-Suchverfahren*, Informatik-Fachberichte 200, Springer Verlag, 1989

[Schaeffer 86] Schaeffer, Jonathan: *Experiments in Search and Knowledge*, Doktorarbeit, University of Waterloo, Nachdruck als: *Technical Report TR 86-12*, Department of Computing Science, University of Alberta, 1986

[Shannon 49] Shannon, Claude: *Programming a Computer for playing Chess*, Philosophical Magazine 41(7), 1949, auch in: *Computer Chess Compendium*, Levy, D. (Hrsg.), S. 2-13, Springer-Verlag, 1988, auch URL http://www.pi.infn.it/~carosi/chess/shannon.txt, CD (Shannon_49.pdf)

[Slate; Atkin 77] Slate, David; Atkin, Larry: *Chess 4.5: The Northwestern University Chess Program*, in: *Chess Skill in Man and Machine,* P.W. Frey (Hrsg.), 1977, auch in: *Computer Chess Compendium,* Levy, D. (Hrsg.), S. 80-104, Springer-Verlag, 1988

[Surratt] Surratt, David: *Chessville Instruction – Perpetual Check*, URL http://www.chessville.com/instruction/Center_Squares/Perpetual_Check/Perpetual_Check.htm

[Tarrasch 31] Tarrasch, Siegbert: *Das Schachspiel*, S. 51, 1. Auflage Berlin, 1931, 7. Auflage Rowohlt Taschenbuch Verlag, Reinbek, 1973

[Schachlexikon 80] Bucher, C.J. (Hrsg.): *Lexikon für Schachfreunde*, S. 28, München, 1980

[Schachmania] Schachmania.de: *Wie kam Schach auf die Maschine? - Geschichte des Computerschach*, URL http://www.schachmania.de/Computer___Schach/computer/computer.html

[Tabibi 02] Tabibi, Omid David: *Verified Nullmove Pruning*, International Computer Games Association Journal, 2002, CD (Tabibi_02.pdf)

[Turing et al 53] Turing, A.M.; Strachey, C.; Bates, M.A.; Bowden, B.V.: *Digital Computers Applied to Games*, in: *Faster Than Thought*, Bowden, B.V. (Hrsg.), 1953, auch in: *Computer Chess Compendium,* Levy, D. (Hrsg.), S. 14-18, Springer-Verlag, 1988

[Wall] Wall, Bill: *Computer Chess History by Bill Wall*, URL http://www.geocities.com/SiliconValley/Lab/7378/comphis.htm

[Weiser 91] Weiser, Mark: *The Computer for the Twenty-First Century*, Scientific American 9/91, S. 94-100, 1991, auch URL http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[Wikipedia Cyber] Wikipedia (Hrsg.): *CDC Cyber 170 series*, URL http://en.wikipedia.org/wiki/CDC_Cyber

[Wikipedia Elo] Wikipedia (Hrsg.): *ELO Wertungssystem*, URL http://de.wikipedia.org/wiki/Elo-Zahl

[Wikipedia Notation] Wikipedia (Hrsg.): *Schach – Algebraische Notation*, URL http://de.wikipedia.org/wiki/Schach#Algebraische_Notation

[Wikipedia Pareto] Wikipedia (Hrsg.): *Pareto-Verteilung*, URL http://de.wikipedia.org/wiki/Pareto-Verteilung

[Xerox PARC] Xerox Corporation: *Palo Alto Research Center*, 2005, URL http://www.parc.xerox.com

[Zanchetta] Zanchetta, Alain: *CEBoard 2.1 Homepage*, 2005, URL http://www.zanchetta.net/CEBoard/

[Zobrist 70] Zobrist, A.L.: *A Hashing Method with Applications for Game Playing*, University of Wisconsin, 1970

[Zweig 74] Zweig, Stefan: *Die Schachnovelle*, S. 21 f., Fischer Taschenbuch Verlag, Frankfurt, 1974

Anhang B: Inhalt der CD-ROM

Der Inhalt der beiliegenden CD-ROM ist folgendermaßen strukturiert:

Ordner	Inhalt
..\Diplomarbeit	Enthält diese Diplomarbeit als PDF-Dokument
..\Eröffnungsbibliothek	Enthält die vollständige Eröffnungsbibliothek, sowie die für Nemesis reduzierte Variante
..\Programm\Nemesis	Enthält das entwickelte Schachprogramm Nemesis 1.0 als eMbedded Visual C++ 3.0 Workspace samt Quellcode-Dateien
..\Programm\Nemesis\ARMRel	Enthält die kompilierte Version von Nemesis
..\Programm\Nemesis\X86EMRel	Enthält die kompilierte Version von Nemesis für den Emulator
..\Programm\NemesisTest	Enthält die Version von Nemesis, die für Teststellungen benutzt wurde
..\Programm\NemesisTest\ARMRel	Enthält die kompilierte Version für Teststellungen
..\Programm\NemesisTest\X86EMRel	Enthält die kompilierte Version für Teststellungen auf dem Emulator
..\Programm\NemesisTestPC	Enthält die Version für Teststellungen auf dem PC als Eclipse 3.0.1 Workspace samt Quellcode-Dateien
..\Programm\NemesisTestPC\Release	Enthält die kompilierte Version für Teststellungen auf dem PC
..\Quellen	Enthält die in Anhang A markierten Quellen
..\Schachregeln	Enthält die vollständigen FIDE-Schachregeln (s. Anhang D)
..\Testergebnisse	Enthält die Ergebnisse der Teststellungen als ASCII-Dateien, die in jede Tabellenkalkulation eingelesen werde können. Hinweise hierzu finden sich ebenfalls in diesem Ordner
..\Testspiele	Enthält die Testspiele als PGN-Dateien. Dieses Format kann z.B. von ChessCaptor 2.21 eingelesen werden (Anhang A [Chess Captor])
..\Teststellungen	Enthält die Teststellungen als EPD-Datei, die ebenfalls von ChessCaptor 2.21 eingelesen werden kann
..\Teststellungen\HTML	Enthält eine Darstellung der Teststellungen im HTML-Format

Anhang C: Testspiele

[White "Nemesis 1.0"]
[Black "tscp181"]
[Result "1-0"]
[TimeControl "40/120"]

1. d4 d5 2. c4 dxc4 3. e4 b5 4. b3 e6 5. bxc4 Bb4+ 6. Ke2 c5 7. a3 bxc4 8. axb4 Qxd4 9. Qxd4 cxd4 10. Na3 Nc6 11. Nxc4 Nxb4 12. Ra4 a5 13. Ke1 Nf6 14. Bd2 Nc2+ 15. Kd1 Nxe4 16. Kxc2 Nxf2 17. Bb4 Nxh1 18. Nd6+ Ke7 19. Nxc8+ Kf6 20. Be7+ Kg6 21. Nd6 Ra7 22. Bd3+ f5 23. Bh4 Kh5 24. Rxd4 Rc7+ 25. Kd1 Rb8 26. Be1 Ra7 27. g4+ fxg4 28. Be4 Rd7 29. Bxh1 Rb6 30. Nf5 Rxd4+ 31. Nxd4 e5 32. Nde2 a4 33. Bg3 a3 34. Nc1 Rb5 35. Nge2 Rc5 36. Be1 Rb5 37. Nc3 Rb2 38. Bg3 Kg5 39. Bd5 e4 40. Nxe4+ Kg6 41. Nd2 Rb5 42. Be4+ Kh6 43. Bd6 Ra5 44. Na2 Ra4 45. Bb4 Ra6 46. Nc4 Re6 47. Nd6 Rf6 48. Ke2 Kg5 49. Bxh7 Rf3 50. Bd3 Rh3 51. Ne4+ Kh6 52. Bd6 Rh5 53. Nf2 g3 54. hxg3 Ra5 55. Ng4+ Kg5 56. Ne3 Kf6 57. Nc2 Rh5 58. Bxa3 Rh3 59. Nc3 Ke5 60. Ne4 Rh2+ 61. Kf3 Ke6 62. Ng5+ Kf6 63. Kg4 g6 64. Bb2+ Ke7 65. Bxg6 Kd6 66. Be5+ Kxe5 67. Nf3+ Kf6 68. Nxh2 Kxg6 69. Kf4 Kf6 70. g4 Kg6 71. g5 Kg7 72. Kf5 Kf7 73. g6+ Ke7 74. Ke5 Ke8 75. Kf6 Kd7 76. g7 Kd6 77. g8=Q Kc7 78. Qa8 Kb6 79. Nf3 Kc7 80. Qa7+ Kc6 81. Qb8 Kd5 82. Qb5+ Kd6 83. Qc4 Kd7 84. Ne5+ Kd8 85. Ke6 Ke8 86. Qc8# 1-0

[White "tscp181"]
[Black "Nemesis 1.0"]
[Result "0-1"]
[TimeControl "40/120"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Bxc6 dxc6 5. d4 exd4 6. Qxd4 Qxd4 7. Nxd4 Nf6 8. Nd2 Bc5 9. N4b3 Ba7 10. f3 Be6 11. Nf1 O-O-O 12. Bg5 Bxb3 13. Bxf6 gxf6 14. axb3 f5 15. Ra4 Rhg8 16. g3 Rge8 17. Nd2 Bd4 18. c3 Be3 19. Nb1 Bb6 20. Nd2 Rd3 21. Ke2 Re3+ 22. Kd1 Rd8 23. Kc1 Red3 24. Nc4 Rxf3 25. Nxb6+ cxb6 26. exf5 Rxf5 27. Rh4 h5 28. Rd1 Rxd1+ 29. Kxd1 Kb8 30. Ke2 a5 31. Kd3 Rf2 32. Rxh5 Rxb2 33. Rh8+ Ka7 34. Rh7 f6 35. Rh6 Rxb3 36. Kc2 Ra3 37. Kb2 Ra4 38. Rxf6 Re4 39. h4 b5 40. Kc2 Re2+ 41. Kd3 Rg2 42. Rf3 a4 43. Kd4 a3 44. h5 a2 45. Rf1 Rxg3 46. h6 Rg4+ 47. Ke5 b4 48. cxb4 Rxb4 49. Ra1 Rh4 50. Rxa2+ Kb8 51. Rf2 Rxh6 52. Rf8+ Kc7 53. Rf7+ Kb6 54. Rf2 c5 55. Kd5 Rh5+ 56. Kc4 Rh4+ 57. Kd5 Rd4+ 58. Ke5 Kc7 59. Rf7+ Rd7 60. Rxd7+ Kxd7 61. Kd5 b6 62. Kc4 Kc6 63. Kd3 Kb5 64. Kc3 c4 65. Kc2 Kb4 66. Kd1 Kb3 67. Ke2 c3 68. Kd1 c2+ 69. Kc1 b5 70. Kd2 Kb2 71. Kd3 b4 72. Kc4 c1=Q+ 73. Kxb4 Ka2 74. Ka4 Qc6+ 75. Kb4 Qc7 76. Ka4 Qb8 77. Ka5 Qd6 78. Ka4 Kb2 79. Ka5 Ka3 80. Kb5 Kb3 81. Ka5 Kc4 82. Ka4 Qb4# 0-1

[White "Nemesis 1.0"]
[Black "tscp181"]
[Result "1-0"]
[TimeControl "40/120"]

1. e4 Nc6 2. d4 e6 3. Bb5 Qh4 4. Bxc6 dxc6 5. Qe2 Bd7 6. Nf3 Qg4 7. g3 O-O-O 8. O-O c5 9. h3 Bb5 10. Qxb5 Qxf3 11. Nd2 Qh5 12. Nb3 a6 13. Qa5 Nf6 14. Bf4 Rd7 15. dxc5 c6 16. e5 Nd5 17. Bc1 Qxe5 18. Re1 Qf5 19. c4 Nf6 20.

Kg2 Be7 21. Bf4 Ne4 22. Rad1 Rhd8 23. Rxd7 Rxd7 24. Be3 e5 25. g4 Qe6 26. f3 Qxc4 27. fxe4 Qxe4+ 28. Kg1 Qf3 29. Bf2 Qxh3 30. Rxe5 Qxg4+ 31. Kf1 f5 32. Re1 g5 33. Qb6 Qc4+ 34. Kg1 h5 35. Qa7 Qg4+ 36. Kh2 h4 37. Qa8+ Kc7 38. Nd4 Bxc5 39. Ne6+ Kb6 40. Nxc5 Qf4+ 41. Kg2 h3+ 42. Kf1 Qc4+ 43. Kg1 h2+ 44. Kh1 Qd5+ 45. Ne4+ Ka5 46. Qh8 Kb5 47. Qc3 b6 48. a4+ Kxa4 49. Bxb6 c5 50. Bxc5 Rb7 51. Ra1+ Kb5 52. Ra5+ Kc6 53. Ba7+ Qc4 54. Qxc4+ Kd7 55. Qf7+ Kd8 56. Rd5+ Kc8 57. Qxf5+ Kc7 58. Qd7# 1-0

[White "tscp181"]
[Black "Nemesis 1.0"]
[Result "1/2-1/2"]
[TimeControl "40/120"]

1. e4 e5 2. Nf3 Bd6 3. Nc3 Nf6 4. Bc4 O-O 5. O-O Nc6 6. d4 exd4 7. Nxd4 Nxd4 8. Qxd4 Re8 9. Bd5 Nxd5 10. exd5 b6 11. Bd2 Ba6 12. Rfe1 Bc5 13. Rxe8+ Qxe8 14. Qg4 Qe7 15. Re1 Qf6 16. Qf4 Bd4 17. Nd1 c6 18. c3 Qxf4 19. Bxf4 Bf6 20. d6 Rf8 21. f3 g5 22. Bd2 Bc4 23. Ne3 Be6 24. Ng4 Bd8 25. Kf2 c5 26. Ne5 f6 27. Nc6 Bxa2 28. Nxd8 Rxd8 29. Ra1 Bd5 30. Rxa7 Bc6 31. f4 g4 32. Be3 Re8 33. b3 Re6 34. Ra6 Rxd6 35. Rxb6 Rd3 36. c4 Rc3 37. g3 Rc2+ 38. Kg1 Kg7 39. Bxc5 Rg2+ 40. Kf1 Rxh2 41. Bf2 Rh1+ 42. Ke2 h5 43. b4 Rc1 44. Kd3 Rd1+ 45. Ke3 Bf3 46. c5 Kf7 47. Rb5 Ra1 48. Ra5 Rb1 49. b5 Rb2 50. Bg1 Re2+ 51. Kd4 Rg2 52. Be3 Rxg3 53. Ra7 Ke6 54. Ra6+ Ke7 55. Ra7 Kd8 56. Ra6 f5 57. Rd6 Kc8 58. Rf6 Be4 59. Rf8+ Kc7 60. b6+ Kb7 61. Rf7 Kc8 62. Rf8+ Kb7 63. Rf7 Kc8 64. Rf8+ Kb7 65. Rf7 Kc8 66. Rf8+ Kb7 1/2-1/2

[White "Nemesis 1.0"]
[Black "GNUChess"]
[Result "0-1"]
[TimeControl "40/120"]

1. d4 Nf6 2. c4 e6 3. Nf3 b6 4. g3 Bb7 5. Bg2 Be7 6. O-O d5 7. Ne5 O-O 8. Nc3 Nbd7 9. Nxd7 Qxd7 10. Bg5 Rad8 11. a4 h6 12. Bf4 c5 13. cxd5 Nxd5 14. Bxd5 exd5 15. dxc5 bxc5 16. Qd3 d4 17. Nb5 a6 18. Na3 g5 19. Bd2 Qxa4 20. f3 Qc6 21. Nc4 Qe6 22. Rf2 Bd5 23. b3 Rfe8 24. Raf1 Bf6 25. Bc1 Qc6 26. Na5 Qb5 27. Nc4 Rb8 28. Qf5 Qc6 29. Bd2 Bg7 30. Na5 Qe6 31. Qc2 Rb5 32. Nc4 Reb8 33. Na5 Bxb3 34. Nxb3 Qxb3 35. Qe4 Qe6 36. Qxe6 fxe6 37. f4 c4 38. Bc1 c3 39. fxg5 hxg5 40. Rf7 Rf8 41. Rxf8+ Bxf8 42. Kg2 c2 43. Rf2 Kg7 44. h4 gxh4 45. gxh4 Be7 46. e4 d3 47. Rf1 Rb1 48. Be3 Rxf1 49. Kxf1 Bb4 50. h5 d2 51. Bxd2 Bxd2 52. Ke2 c1=Q 53. Kd3 a5 54. Ke2 a4 55. Kf2 Qe1+ 56. Kg2 Kh6 57. e5 Bf4 58. Kf3 Qd2 59. Ke4 Qe3# 0-1

[White "GNUChess"]
[Black "Nemesis 1.0"]
[Result "1-0"]
[TimeControl "40/120"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7 6. Bxc6 dxc6 7. d3 Bg4 8. h3 Bxf3 9. Qxf3 O-O 10. Qg3 Qd4 11. Nc3 Bb4 12. Bh6 Nh5 13. Qg4 Bxc3 14. bxc3 Qxc3 15. Qxh5 gxh6 16. Qxh6 f5 17. Qe6+ Rf7 18. Rab1 b5 19. exf5 Qxc2 20. Rb4 Qc3 21. Rg4+ Kf8 22. Qh6+ Ke7 23. Rc1 Qxd3 24. Qe6+ Kf8 25. Qxe5 Qxf5 26. Qh8+ Ke7 27. Qxa8 Qxf2+ 28. Kh1 Rf6 29. Qc8 h5 30. Qxc7+ Kf8 31.

Qg7+ Ke8 32. Re4+ Re6 33. Rxe6+ Kd8 34. Qe7+ Kc8 35. Rexc6+ Kb8 36. Rc8# 1-0

[White "Nemesis 1.0"]
[Black "GNUChess"]
[Result "0-1"]
[TimeControl "40/120"]

1. d4 Nf6 2. c4 e6 3. Nf3 b6 4. g3 Ba6 5. b3 d5 6. cxd5 Qxd5 7. Bg2 c6 8. O-O Qh5 9. Bg5 Ne4 10. Bf4 Bd6 11. Bxd6 Nxd6 12. Nc3 f6 13. Nh4 g5 14. Bf3 Qf7 15. Ng2 f5 16. Ne3 O-O 17. b4 Bb7 18. Qb3 g4 19. Bg2 Qf6 20. h3 gxh3 21. Bxh3 Nd7 22. Rad1 a5 23. bxa5 Rxa5 24. Rd3 Qe7 25. Rfd1 Rf6 26. Bg2 Rh6 27. Kf1 Kh8 28. a4 Rh2 29. Kg1 Rh5 30. Bf3 Rg5 31. Kf1 Rg8 32. Nc4 Nxc4 33. Qxc4 Ba6 34. Qb3 Bxd3 35. Rxd3 Rc8 36. Re3 e5 37. Qc4 e4 38. Kg1 Nf6 39. Kf1 Qd7 40. Nxe4 fxe4 41. Bxe4 Nxe4 42. Rxe4 Qh3+ 43. Kg1 Rh5 44. Rh4 Rxh4 45. gxh4 Rg8+ 46. Qxg8+ Kxg8 47. e3 b5 48. a5 b4 49. a6 b3 50. a7 Qc8 51. f3 b2 52. h5 b1=Q+ 53. Kg2 Qc2+ 54. Kg3 Qc7+ 55. Kh3 Q2h2+ 56. Kg4 Qd7+ 57. Kg5 Qg7+ 58. Kf5 Qxh5+ 59. Ke4 Qgg6+ 60. Kf4 Qhf5# 0-1

[White "GNUChess"]
[Black "Nemesis 1.0"]
[Result "1-0"]
[TimeControl "40/120"]

1. d4 Nf6 2. c4 e6 3. Nf3 b6 4. a3 Bb7 5. Nc3 d5 6. cxd5 exd5 7. g3 Bd6 8. Bh3 O-O 9. Nb5 Re8 10. Nxd6 Qxd6 11. O-O Ba6 12. Re1 Nbd7 13. Bf4 Qe7 14. Qa4 Bc4 15. Qc6 Nf8 16. Ne5 Nh5 17. Bd2 Qf6 18. Qxc7 Ne6 19. Bxe6 Qxe6 20. Nxc4 dxc4 21. Rac1 Rec8 22. Qb7 Nf6 23. e4 Rcb8 24. Qc7 Nxe4 25. Qf4 f5 26. f3 Re8 27. fxe4 fxe4 28. Re3 Qd5 29. Rce1 Qxd4 30. Bc3 Qc5 31. Kh1 Rad8 32. Rxe4 Qc6 33. Kg1 Qc5+ 34. Bd4 Qxd4+ 35. Rxd4 Rxe1+ 36. Kf2 Rde8 37. Rd7 R1e2+ 38. Kf3 R2e7 39. Qxc4+ Kf8 40. Rd2 g6 41. Kg2 Kg7 42. Rf2 Re6 43. Qf4 Kh8 44. b4 R6e7 45. Qd4+ Kg8 46. Qf6 Rd7 47. h4 Rc7 48. Qf4 Rce7 49. Qc4+ Kg7 50. Qd4+ Kg8 51. Qf4 h5 52. Qh6 Re6 53. b5 Rd6 54. Rc2 Re7 55. Rc8+ Kf7 56. Rf8+ Ke6 57. Qxg6+ Kd7 58. Qxh5 Rd4 59. Qf5+ Re6 60. Kh3 Rd1 61. Rf7+ Kd6 62. Qf4+ Re5 63. Qf6+ Kd5 64. Qc6+ Kd4 65. Rd7+ Ke3 66. Rxd1 Re7 67. Qc2 Rd7 68. Re1+ Kd4 69. Re4+ Kd5 70. Qc6# 1-0

Anhang D: FIDE-Schachregeln

Auszug aus Deutscher Schachbund e.V. - FIDE-Schachregeln
Vollständig zu finden unter http://www.schachbund.de/fideregeln/ sowie auf der beiliegenden CD

DIE SPIELREGELN DES WELTSCHACHBUNDES
Beschlossen durch den FIDE-Kongreß 1996
gültig seit 1. Juli 1997

Artikel 1: Wesen und Ziele des Schachspiels

1.1 Das Schach wird zwischen zwei Gegnern gespielt, die abwechselnd Figuren auf einem quadratischen Spielbrett, "Schachbrett" genannt, ziehen. Der Spieler mit den weißen Steinen beginnt die Partie. Ein Spieler ist "am Zug", wenn der Zug seines Gegners abgeschlossen worden. ist.

1.2 Das Ziel eines jeden Spielers ist es, den gegnerischen König so «anzugreifen», daß der Gegner keinen regelgemäßen Zug zur Verfügung hat, der ein «Schlagen» des Königs im folgenden Zug vermeiden würde. Der Spieler, der dies erreicht, hat den Gegner «mattgesetzt» und das Spiel gewonnen. Der Gegner, der mattgesetzt worden ist, hat das Spiel verloren.

1.3 Ist eine Stellung erreicht, in der keinem der beiden Spieler das Mattsetzen mehr möglich ist, ist das Spiel «remis» (unentschieden).

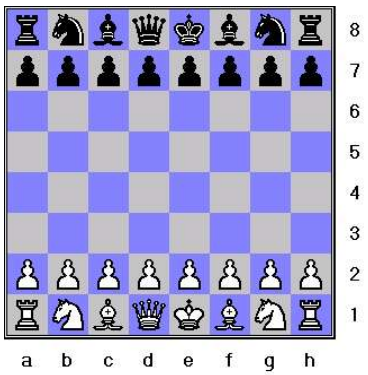
Artikel 2: Die Anfangsstellung der Figuren auf dem Brett

2.1 Das Schachbrett besteht aus einem 8x8-Gitter von 64 gleichgroßen Quadraten, die abwechselnd hell und dunkel sind (die «weißen» und die «schwarzen» Felder). Das Schachbrett wird so zwischen die beiden Spieler gelegt, daß auf der Seite vor einem Spieler das rechte Eckfeld weiß ist.

2.1 Zu Beginn des Spieles hat der eine Spieler 16 helle ("weiße"), der andere 16 dunkle ("schwarze") Figuren. Diese Figuren sind die folgenden:

Ein weißer und ein schwarzer König
Eine weiße und eine schwarze Dame
Zwei weiße und zwei schwarze Türme
Zwei weiße und zwei schwarze Läufer
Zwei weiße und zwei schwarze Springer
Acht weiße und acht schwarze Bauern

2.3 Die Anfangsstellung der Figuren auf dem Schachbrett ist die folgende:



2.4 Die acht senkrechten Spalten von Feldern heißen "Linien", die acht waagerechten Zeilen von Feldern heißen "Reihen". Eine geradlinige Folge von Felder gleicher Farbe, die sich jeweils an den Ecken berühren, heißt "Diagonale".

Artikel 3: Die Gangart der Figuren

3.1 Keine Figur kann auf ein Feld ziehen, das bereits von einer Figur derselben Farbe besetzt ist. Wenn eine Figur auf ein Feld zieht, das von einer gegnerischen Figur besetzt ist, wird letztere geschlagen und als Teil desselben Zuges vom Schachbrett entfernt. Eine Figur greift ein Feld an, wenn diese Figur auf jenem Feld gemäß den Artikeln 3.2 bis 3.5 schlagen könnte.

3.2 a) Die Dame zieht auf ein beliebiges anderes Feld entlang der Linie, Reihe oder einer der Diagonalen, auf welcher sie steht.

b) Der Turm zieht auf ein beliebiges anderes Feld entlang der Linie oder der Reihe, auf welcher er steht.

c) Der Läufer zieht auf ein beliebiges anderes Feld entlang einer der Diagonalen, auf welcher er steht.

3.3 Der Springer zieht auf eines der Felder, die seinem Standfeld am nächsten, aber nicht auf derselben Linie, Reihe oder Diagonalen mit diesem liegen. Er zieht nicht direkt über dazwischenliegende Felder.

3.4 a) Der Bauer zieht vorwärts auf das unbesetzte Feld direkt vor ihm auf derselben Linie, oder

b) er rückt in seinem ersten Zug um zwei Felder entlang derselben Linie vor, sofern beide Felder unbesetzt sind, oder

c) er zieht auf ein von einer gegnerischen Figur besetztes Feld diagonal vor ihm auf einer benachbarten Linie, indem er jene Figur schlägt.

d) Ein Bauer, der ein Feld angreift, das von einem gegnerischen Bauern überschritten worden ist, der vom Ursprungsfeld aus in einem Zug um zwei Felder vorgerückt ist, darf diesen gegnerischen Bauern so schlagen, als ob letzterer nur um ein Feld vorgerückt wäre. Dieses Schlagen darf nur in dem Zug geschehen, der auf ein solches Vorrücken folgt, und wird «Schlagen en passant" genannt.

e) Sobald ein Bauer diejenige Reihe, die am weitesten von seinem Ursprungsfeld entfernt ist, erreicht hat, muß er als Teil desselben Zuges gegen eine Dame, einen Turm, einen Läufer oder einen Springer derselben Farbe ausgetauscht werden. Die Auswahl des Spielers ist nicht auf bereits geschlagene Figuren beschränkt. Dieser Austausch eines Bauern für eine andere Figur wird "Umwandlung" genannt; die Wirkung der neuen Figur tritt sofort ein.

3.5 a) Der König hat zwei verschiedene Gangarten:

I) Er zieht auf ein beliebiges angrenzendes Feld, das nicht von einer oder mehreren gegnerischen Figuren angegriffen wird, oder

II) er «rochiert». Die «Rochade» ist ein Zug des Königs und eines der gleichfarbigen Türme auf derselben Reihe. Sie gilt als ein einziger Zug und wird folgendermaßen ausgeführt: Der König wird von seinem Ursprungsfeld um zwei Felder in Richtung des Turmes hin versetzt; dann wird dieser Turm über den König hinweg auf das Feld, das der König soeben überquert hat, gesetzt.

1) Die Rochade ist regelwidrig,

(A) wenn der König bereits gezogen hat, oder

(B) mit einem Turm, der bereits gezogen hat.

2) Die Rochade ist vorübergehend verhindert:

(A) wenn das Standfeld des Königs oder das Feld, das er überqueren muß, oder sein Zielfeld von einer oder mehreren gegnerischen Figuren angegriffen wird,

(B) wenn sich zwischen dem König und dem Turm, mit dem rochiert werden soll, eine beliebige Figur befindet.

b) Ein König «steht im Schach», wenn er von einer oder mehreren gegnerischen Figuren angegriffen wird, auch wenn diese selbst nicht ziehen können. Das Ansagen eines Schachgebotes ist nicht obligatorisch. Ein Spieler darf keinen Zug machen, der seinen König ins Schach führt oder im Schach stehen läßt.

Artikel 4: Die Ausführung der Züge

4.1 Jeder Zug muß mit einer Hand allein ausgeführt werden.

4.2 Vorausgesetzt, daß er seine Absicht im voraus bekannt gibt (z.B. durch die Ankündigung "j'adoube"), darf der Spieler, der am Zug ist, eine oder mehrere Figuren auf ihren Feldern zurechtrücken.

4.3 Berührt der Spieler, der am Zug ist - außer im Fall des Artikel 4.2 - absichtlich auf dem Brett

a) eine oder mehrere Figuren derselben Farbe, muß er die zuerst berührte Figur, die gezogen oder geschlagen werden kann, ziehen oder schlagen; oder

b) je eine Figur beider Farben, muß er die gegnerische Figur mit seiner Figur schlagen oder, falls dies regelwidrig ist, die erste berührte Figur, die gezogen oder geschalgen werden kann, ziehen oder

schlagen. Fehlen Beweismittel, so gilt, daß die eigene Figur vor der gegnerischen Figur berührt worden ist.

4.4 a) Wenn ein Spieler absichtlich einen Turm und danach seinen König berührt, darf er mit diesem Turm in diesem Zug nicht rochieren; Artikel 4.3 ist dann anzuwenden.

b) Wenn ein Spieler, in der Absicht zu rochieren, seinen König oder König und Turm zugleich berührt, die Rochade aber auf dieser Seite regelwidrig ist, muß der Spieler entweder auf der anderen Seite rochieren, vorausgesetzt, daß die Rochade auf jener Seite zulässig ist, oder seinen König ziehen. Falls der König keinen regelmäßen Zug zur Verfügung hat, darf der Spieler einen beliebigen regelmäßen Zug ausführen.

4.5 Falls keine der berührten Figuren gezogen oder geschlagen werden kann, darf der Spieler einen beliebigen regelgemäßgen Zug ausführen.

4.6 Wenn der Gegner gegen Artikel 4.3 oder 4.4 verstößt, kann der Spieler dies nicht mehr beanstanden, nachdem er selbst absichtlich eine Figur berührt hat.

4.7 Wenn in einem regelgemäßen Zug oder als Teil eines regelgemäßen Zuges eine Figur auf einem Feld losgelassen worden ist, kann sie nicht mehr auf ein anderes Feld gezogen werden. Der Zug gilt als auf dem Brett ausgeführt, wenn alle anwendbaren Anforderungen von Artikel 3 erfüllt worden sind.

Artikel 5: Die beendete Partie

5.1 a) Die Partie ist für den Spieler gewonnen, der den gegnerischen König mit einem regelgemäßen Zug matt gesetzt hat. Damit ist die Partie sofort beendet.

b) Die Partie ist durch den Spieler gewonnen, dessen Gegner erklärt, daß er aufgibt. Damit ist die Partie sofort beendet.

5.2 Die Partie ist unentschieden (remis), wenn der am Zug befindliche Spieler keinen regelgemäßen Zug zur Verfügung hat und sein König nicht im Schach steht. Man sagt dann, «der Spieler wurde patt gesetzt». Damit ist die Partie sofort beendet.

5.3 Die Partie ist remis durch eine von beiden Spielern während der Partie getroffene Übereinkunft. Damit ist die Partie sofort beendet.

5.4 Die Partie darf remis gegeben werden, falls die identische Stellung zum dritten Mal auf dem Brett entstanden ist.

5.5 Die Partie darf remis gegeben werden, falls die letzten 50 aufeinanderfolgenden Züge von jedem Spieler gemacht worden sind, ohne daß irgendein Bauer gezogen oder irgendeine Figur geschlagen worden ist.

[...]

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4), §24(4) bzw. §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. Oktober 2005 _____

Ort, Datum Unterschrift