Berufsakademie Stuttgart
Staatliche Studienakademie

08. Juli 2003

Studienarbeit

BaChess

Entwurf und Implementierung einer X/Winboard kompatiblen Schachengine

Clemens Kutz
Michael Schlagmüller

Kurs: TIT-IN 2000

Betreuung: Prof. Dr. Karl Friedrich Gebhardt

1 Vorwort

Diese Studienarbeit ist im Rahmen der Studienarbeit von Clemens Kutz und Michael Schlagmüller an der BA-Stuttgart entstanden.
Im Bereich der künstlichen Intelligenz und der Schachprogrammierung sind bereits viele hervorragende Arbeiten entstanden und im Internet publiziert worden, so dass man das Rad nicht neu erfinden muss, für diese Aufgabe. Trotz der ganzen Informationen ist es uns nur mit viel Mühe gelungen, ein funktionierendes und gut spielendes Schachprogramm zu entwickeln. Folgende Weisheit haben wir im Bezug auf das Erstellen eines Schachprogramms gefunden, die wir voll unterstützen können: „Es ist keine Kunst, ein Schachprogramm zu programmieren, sondern es zu debuggen“

Im folgenden Text wird folgende Notation verwendet:
- GROSSE Wörter bedeuten ein Name eines Schachprogramms
- *kursive* Wörter sind Begriffe, die im Glossar erklärt werden

Des weiteren besitzt diese Arbeit ein Quellenverzeichnis, auf das aber nicht direkt im Text hingewiesen wird.

Bedanken möchte ich mich bei folgenden Personen:

- Aske Plaat für seine ausführlichen Beschreibungen der Suchalgorithmen, die wir leider nicht 100% fehlerfrei umsetzten konnten (siehe Quellen)
- François-Dominic Laramée für seine hervorragende Arbeit mit dem Artikel über „Chess Programming“ – leider auch Fehlerhaft, aber trotzdem sehr gut (siehe Quellen)
- Alejandro Dubrovsky mit SMALL POTATOE, dessen Suchalgorithmus bei uns Verwendung findet
- meiner Mutter, für die Arbeit, die sie sich gemacht hat diesen Text zu lesen und zu korrigieren
- Ina, die Clemens eine Zeit lang entbehren musste
- Herrn Gebhardt, der uns im Rahmen dieser Arbeit mehrere Kisten Bier gesponsert hat – Aufgrund einer verlorenen Wette bezüglich Java und C++ und diesem Schachprogramm
- Michael Schlagmüller, der den harten Kampf gegen den inneren Schweinehund gewonnen hat und wochenlang Algorithmen gedebugt hat und umgecodet hat
- Clemens Kutz, für seine hervorragende Leistung an dieser Arbeit und dem ständigen Hinweis, dass es wichtig ist seinen Code zu dokumentieren und nicht nur einfach guten Code zu schreiben, sonst würde der Code jetzt ganz anders aussehen
- Gott, der uns die technischen Möglichkeiten gegeben hat, uns dazu befähigt hat dies alles zu tun.
  „Größe und Reichtum mag ein Mensch gewinnen;
  aber wenn er keine Einsicht hat, geht er zugrunde wie Vieh.“ (Psalm 49, 21)

2 Einleitung

Das Programmieren eines Schachprogrammes ist eine der Standardaufgaben für Informatikstudenten. Es gibt dazu sehr ausführliche Dokumente wie Bücher oder Internetseiten. Trotzdem ist es immer wieder neu eine Herausforderung den Computer intelligenter zu machen als den Durchschnittsmensch beim Schachspielen – und das ist keine einfache Aufgabe.
Im professionellen Bereich sind wir mittlerweile soweit, dass man die Personen, die die besten Schachprogramme regelmäßig schlagen wohl an zwei Händen abzählen kann. Die Programme sind durch die immer fortschreitende Technik und dem verbesserten Programmcode wesentlich stärker geworden. Das menschliche Schachspielen hingegen verändert sich kaum. Somit ist es absehbar, dass der Computer den Menschen in dieser Disziplin schlagen wird. Unserer Prognose nach wird dies keine 10 Jahre mehr dauern.
Deshalb hier ein kleiner Rückblick auf die (Computer-)Schachgeschichte:
Das Schachspiel ist nicht nur von einem Menschen entstanden. Es ist ein so geniales Spiel, dass es Zeit gebraucht hat sich zu entwickeln. Die heutigen Forschungen ergeben, dass das Schachspiel wahrscheinlich im 6. Jahrhundert nach Christus in Indien entstanden ist. Die Araber lernten das Spiel von den Indern kennen, als sie im 7. Jahrhundert deren Land eroberten. Schon in dieser Zeit entstanden Werke über das Schachspiel wie über Schachprobleme oder schachtheoretische Fragen. Die ersten wesentlichen Regeländerungen wurden bei der Übernahme des Spiels von den Indern zu den Arabern festgestellt. Durch die Araber kam das Spiel zunächst nach Nordafrika und von dort ist es vermutlich gegen Ende des 8. Jahrhunderts nach Spanien gebracht worden. Dann nahm es schnell seinen Weg in das übrige Europa, wo es bereits im 11. Jahrhundert allgemein bekannt war. Dort erreichte das Schachspiel alle Kreise der Bevölkerung. Es gehörte zum guten Ton Schach spielen zu können, sowohl bei Männern als auch bei Frauen. Die modernen Regeln wurden zwischen 1400 und 1475 eingeführt. Im Jahr 1851 wurde das erste internationale Schachturnier in London ausgetragen. Damit begann die Ära der offenen Schachkämpfe. In der kommenden Zeit wurde das Schachspiel (nicht die Regeln, sondern die Art zu spielen) öfters durch neue Theorien verändert, bis hin zum heutigen Tag. Es wurde hervorragende Literatur von den Großmeistern des Schachs herausgebracht. Nur um ein paar Großmeister zu nennen: Fischer, Spassky, Karpow, Kasparov, ...
Seit 1940 wird ernsthaft an der Computerschach-Problematik gearbeitet. Zuerst wurden Theorien entwickelt, die aber noch nicht umgesetzt werden konnten, da die verfügbaren Rechner zu schlecht waren. Erst mit der Zeit wurde der Computer zum ernsthaften Gegner des Menschen. 1977 erschien der erste käufliche Schachcomputer auf dem Markt. Seine Spielstärke war allerdings sehr gering. Dies änderte sich rasant. 1983 erreichten die Computer Vereinsniveau. Im Jahr 1994 gewann ein 486/33 fast ein weltklasse besetztes Blitzschachturnier (Kasparov, Anand, Short, Kranik, Hübner, ...) mit *FRITZ* 3. Er erreichte die meisten Siege im Turnier, den Stichkampf mit Garry Kasparov verlor er allerdings. Das wohl bekannteste Turnier zwischen Mensch und Maschine ist der Kampf zwischen *DEEP BLUE* und Gary Kasparov. Die Hinrunde 1996 gewann Kasparov gegen die 200 parallel geschalteten IBM Rechner (RS/6000 Workstations). Die Rückrunde 1997 verlor er allerdings gegen das verbesserte Programm und der verdoppelten Taktfrequenz der Rechner. Dazu ist allerdings zu sagen, dass Kasparov ungewöhnlich schlecht spielte und somit dem Computer Chancen eröffnete, die dieser nutzte. Zudem waren die Bedingungen für Kasparov beim Turnier schlecht (wenig Vorbereitungszeit, ...). Ein weiteres bekanntes Duell fand 2002 zwischen Weltmeister Kramnik und *DEEP FRITZ* statt. *DEEP FRITZ* arbeitete auf einem Rechner mit 8 Xeon Prozessoren, die mit 900 MHz getaktet waren. Kramnik schaffte es zur Halbzeit mit 3-1 zu führen, verlor allerdings einige der folgenden Partien und das Duell endete 4-4 unentschieden.
Somit ist bisher noch nichts entschieden. Wenn die Großmeister genügend Zeit haben, ihr gegnerisches Computerprogramm zu studieren können immer Schwächen in der Software

entdeckt werden, die zum Sieg des Menschen führen. Allerdings hat dies wenig mit dem klassischen Schachspiel zu tun. Man geht beispielsweise beim Computerschach davon aus, dass es wichtig ist schnell die Damen auszutauschen, da der Computer zu mächtig mit ihr spielen kann. Es existieren weitere Regeln dieser Art, die jedoch abhängig vom Computerprogramm sind. Jetzt aber zur Entwicklung dieses hier vorgestellten Schachprogramms.

3 Inhaltsverzeichnis

4 KI & Schach (allgemeine Theorien)

Schach ist neben Spielen wie Backgammon, Dame, *Go* und vielen weiteren ein Spiel, bei dem man mit einem Blick auf das Spielfeld, eine Aussage darüber treffen kann, wie es um die Spielsituation steht, da alle Informationen über das Spiel offenliegen. Im Gegensatz dazu steht beispielsweise das Pokerspiel, bei dem jeder der Spieler seine Karte verdeckt hält.
Für die Spiele der ersteren Gruppe gibt es weit fortgeschrittene Programmieransätze um eine KI zu entwickeln. Im Folgenden wird eine allgemeine Übersicht über einen Ansatz am Beispiel von Schach aufgezeigt:

Ein Schachprogramm muss folgende Komponenten besitzen:

- eine Möglichkeit, das Spielbrett im Speicher abzubilden, um Spielsituation festhalten und Algorithmen auf diese anwenden zu können
- Spielregeln, die einerseits dazu genutzt werden alle Spielmöglichkeiten des Computers zu berechnen, andererseits um die Spielzüge des Menschen zu überprüfen
- eine Technik, um unter den fast unendlichen vielen Zugkombinationen eine möglichst sinnvolle auszuwählen
- dazu müssen das Spielbrett und die Spielsituation bewertet werden. Mit der Hilfe einer Bewertung kann der ‚beste' Zug ermittelt werden
- des weiteren wird eine Oberfläche benötigt

4.1 Darstellung des Spielbretts

Vor einigen Jahren, als Programme noch auf geringe Speichernutzung optimiert werden mussten, wählten Programmierer eine einfache und effiziente Art, das Schachbrett abzubilden: eine Feld von 8 x 8 Einträgen, in dem jedes der 8 x 8 Bytes (64 Spielfelder) die Spielfigur, die auf ihm steht, speichert (z.B. 0: leeres Feld, 1: weißer König, 2: ...)
In den folgenden Jahren nahm die Problematik des Speichermangels durch den Fortschritt der Technik ab. Mit dem Einsatz von 64-Bit Rechnern bot es sich an, das Spielfeld in *BitBoards* zu speichern. Alle Informationen wurden in 64-Bit Worten hinterlegt (jedes Bit ein Spielfeld): beispielsweise die Belegung der schwarzen Bauern auf dem Spielfeld, oder alle möglichen Zugmöglichkeiten die ein schwarzer Läufer auf dem Feld b3 hat. Dies hat den Vorteil, dass 64-Bit Rechner Operationen auf das aktuelle Spielfeld in nur einem einzigen Rechenzyklus abarbeiten können, was die Performance deutlich steigert.

4.2 Berechnung der Zugmöglichkeiten

Die Spielregeln des Schachs bestimmen die möglichen Züge der beiden Spieler in einer bestimmten Spielsituation. Es gibt Spiele, bei denen die Berechnung der möglichen Züge trivial ist: bei Tic-Tac-Toe zum Beispiel darf man nur leere Felder belegen. Beim Schach hingegen besitzt jede Figur andere Möglichkeiten sich auf dem Schachbrett fortzubewegen. Es gibt Sonderregeln, wie das *en passant* bei Bauern (Schlagen im Vorübergehen), die *Rochade* und vieles mehr.
Diese vielfältigen Möglichkeiten machen eine Berechnung aller möglicher Spielzüge schwierig und zeitaufwendig. Die Erstellung der Zugmöglichkeiten stellt ein Hauptteil eines Schachprogrammes dar. Deshalb werden nicht alle Zugmöglichkeiten zur Laufzeit neu be-

stimmt, sondern es wird auf vorberechnete Datenbanken zurückgegriffen, um diesen Vorgang und damit das Schachprogramm zu beschleunigen.

4.3 Suchalgorithmen

Für den Computer ist es schwierig, die aktuelle Spielsituation zu bewerten, was jedoch nötig ist, um Vorhersagen treffen zu können, welcher Zug sinnvoll ist und welcher nicht. Ein einfacherer Ansatz eine gute Zugkombination zu entdecken besteht darin, einige Zugkombinationen zu durchlaufen und die Konsequenzen, die sich aus diesen Zugmöglichkeiten ergeben, zu bewerten. Dieses wird mit Hilfe eines *MiniMax-Algorithmus* bestimmt, der die Basis eines jeden Schachprogrammes darstellt.
Leider ist die Komplexität des *MiniMax* $O(B^N)$, wobei B (*Branching-Faktor*: Verzweigungsfaktor) die Anzahl der im Schnitt möglichen Züge ist und N (Zugtiefe) die Anzahl der Halbzüge ist, die vorberechnet werden. Ein Halbzug ist der Zug von nur eine Seite. (z.B. weiß). Die Komplexität steigt also exponentiell mit der Suchtiefe, so dass ein enormer Aufwand betrieben werden muss, diesen Aufwand zu minimieren
Ein weiteres großes Problem von Schachprogrammen stellt der *Horizonteffekt* dar. Folgendes Beispiel beschreibt die Problematik:
Man stelle sich vor, dass das Schachprogramm 8 Halbzüge im Voraus berechnet. Es stellt beim 6. Zug fest, dass er seine Dame verlieren wird, außer es opfert einen Läufer, um die Dame zu retten. Doch dieser Zug verhindert in dieser Spielsituation nicht das Schlagen der Dame, sondern zögert es nur bis zum 10. Halbzug hinaus. Da das Programm aber nur bis zum 8. rechnet denkt es, die Dame sei sicher und opfert sinnlos einen Läufer. Um dieser Problematik zu begegnen wurden verschiedene Techniken entwickelt. Dir *Ruhesuche* (q*uiescence search)* und die *vereinzelte Ausdehnung* (*singular extension)* von *DEEP BLUE* sind die bekanntesten.

4.4 Bewertung

Um entscheiden zu können, ob ein Zug sinnvoll ist oder nicht, muss das Programm nach der Ausführung einer Zugkombination das neu entstandene Spielbrett bewerten können. Die Bewertung ist maßgeblich abhängig von den Spielregeln des Spiels. Beim Schach spielt das auf dem Spielbrett verfügbare Material eine große Rolle. Der Vorteil von nur einem Bauer kann spielentscheidend sein, vor allem bei Spielern, die sehr gut spielen können. Dies steht im Gegensatz zu anderen Spielen wie z.B. *Go*, bei dem bis zum letzten Zug nicht gesagt werden kann, wer gewinnen wird. Mit dem letzten Zug kann das komplette Spiel unentschieden werden.
Die Entwicklung eines Algorithmus zur Bewertung des Schachbretts ist kompliziert. Dazu kommt, dass dieser Algorithmus oft aufgerufen wird, um die vielen möglichen Züge zu bewerten. Bei zu komplexen Algorithmen geht zuviel Performance verloren, auf Kosten der Suchtiefe. Somit muss ein Kompromiss zwischen diesen beiden Aspekten gefunden werden.

5 Datenstrukturen

In den letzten 30 Jahren hat sich an den grundlegenden Ideen, die das Schachspiel wiedergeben, nichts geändert. Die damals entwickelten Strukturen waren gut durchdacht und bis heute erfolgreich.
Zwei der in den folgenden Abschnitten erklärten drei Datenstrukturen werden dazu verwendet, sich schneller durch den Suchbaum zu bewegen. Die dritte Datenstruktur dient zur beschleunigten Erstellung der Zugmöglichkeiten.

5.1 Darstellung des Spielbretts

In den 70ger Jahren waren die Computer mit wenig Arbeitsspeicher ausgestattet. Aus dieser Tatsache heraus entwickelte sich die Darstellung des Schachbretts als Feld von 64 Bytes, bei dem jedes Byte eine Spielfigur repräsentiert. Zudem müssen zusätzlich Informationen über Sonderregeln gespeichert werden, wie *en passent* Bauern, ob eine *Rochade* noch erlaubt ist, usw. Diese besonderen Ausnahmen werden nicht direkt in die Darstellung des Spielbretts integriert sondern in eigenen Datenstrukturen abgelegt und damit auch gesondert behandelt.

Einige wenige Weiterentwicklungen der Ursprünglichen Ideen sind erwähnenswert:

- Das Schachprogramm SARGON erweiterte die 8 x 8 Spielfeldrepräsentation durch zwei Felder in jeder Richtung, so dass ein 12x12 Feld entstand. Die neu hinzugekommenen Felder wurden als ungültig markiert. Der Vorteil einer solchen Spielfeldrepräsentation besteht darin, dass die Erstellung der Zugmöglichkeiten wesentlich vereinfacht und damit beschleunigt werden kann. Es muss nicht mehr im Vorfeld überprüft werden, ob eine Zug einer Spielfigur auf ein ungültiges Feld führen würde. Der Zug wird einfach ausgeführt und danach geprüft, ob das Zielfeld ein gültiges Feld ist. Beispielsweise zieht der Turm solange in eine Richtung, bis er ein ungültiges Feld erreicht. Die zweite Reihe an hinzugefügten Feldern ist für die Springer notwendig, da diese am Rand stehen und die erste Reihe an illegalen Feldern überspringen können.
- MYCHESS ging einen anderen Weg. Es verwendete statt der 64 Spielfelder die Abbildung der 24 Spielsteine auf ein Feld von 24 Bytes, welches die Position der Figur enthält, bzw. einen bestimmten Wert, falls die Figur bereits geschlagen wurde. Diese Abbildung bringt einige Probleme mit sich, wie beispielsweise die Verwandlung eines Bauern beim Erreichen der gegnerischen Offiziersreihe mit einer Figur, die noch nicht geschlagen wurde.

Heutzutage werden durch den im Vergleich unbegrenzten verfügbaren Arbeitsspeicher verschwenderischere Strukturen für die Darstellung des Spielbretts gewählt. Die ursprünglichen Ideen findet man höchstens auf Palms oder Handys wieder, bei denen noch auf Speicherverbrauch geachtet werden muss.
Es gibt kaum eine effizientere Repräsentation des Spielfeldes, als die klassische Variante des 1-Feld 1-Byte. 1960 wurde allerdings eine neue Möglichkeit von den KAISSA Entwicklern gefunden: das *BitBoard*.
KAISSA lief auf einem Großrechner, der mit einem 64-Bit Prozessor ausgestattet war. Diese 64 Bits wurden dazu genutzt das ganze Spielfeld abzudecken (jedes Bit ein Feld). Um die verschiedenen Spielfiguren zu speichern, werden mehrere *BitBoards* parallel verwendet. Eines, das die weißen Bauer abdeckt, eines, für die schwarzen Bauern. Ein weiteres, in dem alle Felder gespeichert sind, auf denen weiße Figuren stehen, ...

Üblicherweise sind es 12 *BitBoards*, die das komplette Spielfeld repräsentieren: jeweils ein *BitBoard* für jede Spielfigur und Farbe. Zusätzlich werden zwei weitere *BitBoards* verwendet um die gesamten Spielfelder speichern zu können, die die jeweilige Partei (weiß / schwarz) belegen. Die vorteilhafte Implementierung von *Bitboards* vereinfacht zum Beispiel das Problem um feststellen zu können, ob die Dame des weißen Spielers den schwarzen König schach stellt. Bei einer einfachen 64 Felder Darstellung würde sich folgende Lösung anbieten:

- suche die Position der weißen Dame mit Hilfe einer linearen Suche über die Felder. Im ungeschicktesten Fall benötigt dies 2 x 64 Schritte (1. laden und 2. überprüfen von jedem Feld).
- untersuche die Felder, auf die die weiße Dame ziehen kann, in alle acht möglichen Zugrichtungen der Dame, bis entweder der König gefunden wird oder keine weiteren Zugmöglichkeiten existieren.

Diese Suche würde recht lange dauern, vor allem wenn die Dame in einem hinteren Feld der Datenstruktur liegt und kein Schach vorliegt. Letzteres ist fast immer der Fall.

Mit einer *BitBoard* Repräsentation würde sich folgendes ergeben:

- lade das *BitBoard* der Dame
- dieses *BitBoard* wird als Index für ein in der Datenbank gespeichertes *BitBoard* verwendet, das die Felder repräsentiert, die durch diese Dame bedroht werden.
- logisches verunden mit dem *BitBoard* der Position des schwarzen Königs liefert das Ergebnis

Falls diese ungleich null ist, bedroht die Dame den König. Geht man davon aus, dass das *BitBoard* der Dame, das ihre Zugmöglichkeiten wiedergibt, im Cache steht, kann die ganze Berechnung innerhalb von 3-4 Prozessorzyklen berechnet werden !

Ein anders Beispiel: Um alle Zugmöglichkeiten der weißen Springer zu bestimmen werden die beiden Angriffs-*BitBoards* der beiden Springer verodert und anschließend mit dem negierten *BitBoard* für alle weißen Figuren verundet. Dies ergibt alle Zugmöglichkeiten für die weißen Springer.

Es kann allerdings der berechtigte Einwand erbracht werden, dass die meisten PCs heutzutage keine 64 Bit Rechnerarchitektur besitzen. Durch Aufteilen der 64 Bit Operationen in mehrere 32 Bit Operationen geht natürlich Performance verloren, doch ist dies immer noch schneller als die alte 1 Feld –1 Byte Darstellung. Zudem ist es absehbar, dass in den nächsten Jahren auch der PC Bereich auf eine 64 Bit Architektur umsteigen wird.

5.2 Transpositionsverzeichnis

(Umstellungsverzeichnis)

Beim Schach gibt es oft verschiede Möglichkeiten ein und dieselbe Spielfeldstellung zu erreichen. Es ist zum Beispiel egal, ob man 1. a4 ... 2. g4 oder 1. g4 ... 2. a4 spielt; das Resultat ist das gleiche. Um doppelten Bewertungsaufwand für beide Zugzweige zu vermeiden wurde das *Transpositionsverzeichnis* entwickelt.
Das *Transpositionsverzeichnis* stellt ein Speicher für vergangene Suchergebnisse dar, auf das mit Hilfe von *Hash-Werten* über der Spielbrettstellung zugegriffen wird. Sobald eine Spielsituation untersucht wird, wird dieses Ergebnis (z.B.: Materialbewertung, Suchtiefe, bester Zug, usw.) im Verzeichnis abgelegt. Falls später weitere Stellungen untersucht werden müssen

wird zuerst im *Transpositionsverzeichnis* abgeprüft, ob bereits eine nutzbare Bewertung vorhanden ist, die weitere Untersuchungen vereinfacht.
Daraus ergeben sich folgende Vorteile:

- Geschwindigkeit: In Spielsituationen bei denen leicht verschiedene Zugkombinationen zur selbe Situation führen (wie z.B. im Endspiel, wenn nur noch wenige Figuren vorhanden sind), befinden sich schnell die meisten Ergebnisse im *Transpositionsverzeichnis*, so dass 90% der nötigen Bewertungsberechnungen direkt aus dem Verzeichnis geholt werden können, ohne die Stellung neu bewerten zu müssen.
- höhere Genauigkeit: Angenommen die künstliche Intelligenz soll mit einer Suchtiefe von 4 (2 Züge auf jeder Seite) eine Spielsituation bewerten und sie findet im *Transpositionsverzeichnis* bereits eine Bewertung für eine höhere Suchtiefe von 6 Halbzügen für die gleiche Stellung der Figuren. Dadurch erreicht man von vornherein eine höhere Genauigkeit, ohne eine zeitaufwendige Bewertung der Situation vornehmen zu müssen

Der einzige Nachteil, der sich durch die Verwendung des *Transpositionverzeichnisses* ergibt ist die Tatsache, dass möglichst viele Stellungen gespeichert werden müssen und somit ein nicht zu unterschätzender Speicherverbrauch vorliegt. Bei 16 Bytes für einen *Transpositionseintrag* und einer Feldgröße von mindestens einer Millionen Stellungen ergeben die knapp 16 MB. Bei Architekturen wie beim *Palm* oder ähnliches müssen hier deutliche Abstriche gemacht werden.

5.2.1 Weitere Möglichkeiten eines Transpositionsverzeichnisses

Es können neben der kompletten Bewertung der Spielsituation weitere *Transpositionsverzeichnisse* zur weiteren Beschleunigung von rechenintensiven Programmabschnitten verwendet werden. Beispielsweise durch die Speicherung der Bauernstruktur und seiner Bewertung. Es gibt relativ wenige Möglichkeiten für die Bauern sich fortzubewegen, so dass die Situationen Platz sparend gespeichert werden können. Hinterlegt ist die Bewertung der Bauernstruktur, die sonst immer zeitaufwendig neuberechnet werden muss. Eine andere Möglichkeit ist die Materialverteilung in einem *Transpositionsverzeichnis* zu hinterlegen, die sich nur ändert, wenn eine Figur geschlagen wird.
Diese Optimierungen verlieren an Wert, wenn man sieht wie schnell die heutigen Rechner geworden sind. Doch die grundlegende Idee ist wichtig: Speicherung von alten Bewertungen und das Nutzen von Berechnungen, die bereits im voraus ausgeführt wurden. Mit diesen mitteln kann Rechenzeit eingespart werden – auf Kosten von Arbeitsspeicher.

5.3 Berechnung von Hashschlüsseln über das Spielbrett

Das oben vorgestellte *Trasponitionsverzeichnis* wird meist über *Hashtabellen* implementiert, wodurch eine gute und schnelle Erstellung von *Hashschlüsseln* gefordert wird.

Im Jahr 1970 stellte Zorbist folgende Idee zur Generierung des *Hashschlüssels* auf:

- Im Vorfeld werden 12x64 N-Bit Zufallszahlen erstellt (bei denen das *Transpositionsverzeichnis* 2^N Eintrage besitzt) und in einer Tabelle abgelegt. Jede Zufallszahl bezieht sich auf eine bestimmte Spielfigur auf einem bestimmten Spielfeld (beispielsweise der schwarze Turm auf h4). Ein leeres Feld ist ein leeres Wort.
- Jeder neue Hashvorgang beginnt mit einem leeren Wort.

- Jedes Spielfeld wird gesucht und der *Hashschlüssel* mit dem jeweiligen Eintrag für das Feld mit seiner Figur verX-odert.

Der Vorteil dieser Generierung von *Hashschlüsseln* ist der, dass nach einem Zug sehr einfach der neue *Hashschlüssel* berechnet werden kann, ohne jedes Feld erneut untersuchen zu müssen. Durch die logischen Eigenschaften des X-Oders genügt es, die *Hashschlüssel* der veränderten Figuren auf den *Hashwert* des alten anzuwenden. Ein Beispiel: Der weiße Turm (h1) schlägt einen schwarzen Bauern (h4). Um den neuen *Hashwert* zu berechnen, muss erneut das X-oder mit dem Wert für „weißer Turm auf h1“ angewendet werden (wegnehmen des Turms von h1). Es folgen „schwarzer Bauer auf h4“ (schlagen des Bauern), sowie „weißer Turm auf h4“ (neue Position des Turms).
Genau dieselbe Operation wird mit einem zweiten Satz von Zufallszahlen gemacht, um einen *Hashlock* (Einzigartigkeit des *Haschschlüssels* mittels *doppeltem Hashing*) zu erlangen. Dieser Wert wird dazu verwendet *Hashkollisionen* zu erkennen und falsche Interpretationen bei Kollisionen zu vermeiden. Es ist bei geeignetem Algorithmus und geeigneten Zufallszahlen sehr sehr unwahrscheinlich, dass sowohl der *Hashwert*, als auch der *Hashlock* für zwei verschiede Spielbrettstellungen übereinstimmen.

5.4 Historytabelle (Speicherung von vergangen Zügen)

Eine weitere für die Beschleunigung notwendige Datenstruktur ist die *Historytabelle*. Sie wird dazu verwendet möglichst früh einen guten bereits in anderer Situation ausgeführten Zug erneut anzuwenden, in der Hoffnung, dass er wieder gut ist. Eine ausführliche Erklärung folgt im Kapitel über die Suchalgorithmen.

5.5 Vorberechnung von Zugmöglichkeiten

Die Berechnung der möglichen Züge für einen Spieler bildet zusammen mit der Bewertung des Spielbretts den größten Zeitaufwand bei einem Schachprogramm. Deshalb werden Vorberechnungen für die Zugerstellung verwendet, um das ganze Schachprogramm zu beschleunigen:

- für die Erstellung der Zugmöglichkeiten ist es egal, welche Farbe die Figuren haben (wenn man von den Bauern absieht, die genau die entgegengesetzte Bewegungsrichtung haben)
- es gibt 64 (Felder) x 5 (*Offizier*figuren) = 320 Möglichkeiten der *Offiziere*, zu ziehen. 48 mögliche Felder, auf denen ein weißer Bauer einer Farbe stehen kann (sie können nicht zurück gehen, und wenn sie die letzte Reihe erreichen werden sie zum *Offizier*), und 48 Felder für die schwarzen Bauern
- jede Spielfigur hat Zugrichtungen, in die sie ziehen kann. Eine Dame auf h3 hat z.B. eine nördliche Zugrichtung, eine südliche, ...
- jede Spielfigur auf einem bestimmten Feld hat evtl. nur noch eingeschränkte Zugrichtungen zur Auswahl. Ein König in der Mitte hätte alle 8 Richtungen zu Verfügung, wohingegen ein Läufer in einer Ecke nur eine mögliche Zugrichtung besitzt.
- vor dem Spiel wird eine Datenbank mit allen möglichen Zugrichtungen, mit allen möglichen Spielfiguren, auf allen möglichen Feldern berechnet. (dabei werden die berechneten Zugmöglichkeiten nicht durch Spielsteine, die im Weg stehen könnten begrenzt, sondern nur vom Spielfeldrand)

- sollen nun die Zugmöglichkeiten für eine Spielfigur im Spiel berechnet werden, werden alle Zugrichtungen aus der Datenbank entnommen und solange verfolgt, bis auf eine Spielfigur gestoßen wird. Falls sie eine gegnerische Spielfigur ist kann sie geschlagen werden und der Zug gehört noch zu den erlaubten, falls nicht, wird er verworfen.

Mit einer sinnvoll aufgebauten Datenbank wird die Erstellung der Zugmöglichkeiten somit auf ein lineares Nachschauen in der Datenbank reduziert, bei der keine Berechnungen mehr stattfinden müssen. Die ganze Datenbank verbraucht dabei nur einige Kilobyte, was im Vergleich zur *Transpositionstabelle* vernachlässigbar ist.

6 Berechnung der Zugmöglichkeiten

Beim Schach hat ein Spieler oft mehr als 30 mögliche Züge die er ziehen kann, von denen er einen auswählen muss. Dies stellt den Computer vor ein Problem, da es meist wenige gute Züge gibt, einige schlechte und viele mit verheerenden Folgen. Einem menschlichen Spieler, der das Schachspielen beherrscht, fällt es nicht schwer viele der Züge auszuschließen und einen Sinn in seinen Zügen hineinzulegen. Selbst ein Anfänger weiß, dass man seine Figuren nicht ungedeckt einfach aufs Feld ziehen sollte, ohne damit einen Plan zu verfolgen.
Diese Intuition des Menschen und die daraus resultierende Strategie kann nur unter sehr großen Anstrengungen einem Computer beigebracht werden. Deshalb haben die meisten Programme diese Möglichkeit, den Schachcomputer intelligent zu machen, verworfen (bis aus ein paar wenige – beispielsweise Hans Berliner mit HITECH und seinen Weiterentwicklungen). Die andere Möglichkeit ist rohe Gewalt anzuwenden und einfach tief zu rechnen: es ist nicht notwendig dass man mit einem genauen Plan seine Züge auswählt, wenn man schnell genug viele Züge berechnen kann und man die Konsequenzen, die daraus entstehen, bewertet. Durch die Suchtiefe entsteht automatisch eine gute Zugkombination. Deshalb muss die Erstellung der Zugmöglichkeiten und die Bewertung der ausgeführten Züge schnell sein, dass möglichst tief gerechnet werden kann. Es kann dadurch der Eindruck erweckt werden, dass der Computer wirklich einen Plan mit seinen Zügen verfolgt, was aber nicht wahr ist, da er nur Konsequenzen von Zügen bewertet.

Folgende Möglichkeiten ergeben sich für die Berechung der Zugmöglichkeiten:
- selektive Zugerstellung: zuerst wird das Spielfeld untersucht und einige wenige sinnvolle Züge herausgepickt – die anderen werden verworfen
- inkrementelle Zugerstellung: einige wenige Züge werden erstellt in der Hoffnung, dass ein genialer Zug unter ihnen ist, so dass eine weitere Suche unnötig wird
- komplette Zugerstellung: alle möglichen Züge werden erstellt in der Hoffung, dass viele der Züge bereits im *Transpositionsverzeichnis* bewertet sind und somit keine Notwenigkeit besteht, diese ganzen Züge auch auszuführen und zu bewerten

Die selektive Zugerstellung (und der dazugehörige Suchalgorithmus ‚*forward pruning*' (vorwärts gerichtete Beschneidung) ist mit der Weiterentwicklung der Computertechnik bereits Mitte der 70ger Jahre unwichtig geworden. Die inkrementelle Zugerstellung braucht einen höheren Aufwand für die Erstellung aller Zugkominationen, als es der Algorithmus für die komplette Zugerstellung benötigt. Da es aber evtl. durch einen bereits gefundenen und ausgewerteten Zug unnötig ist, weitere Züge zu erstellen kann er auch schneller sein. Bei der kompletten Zugerstellung wird der Algorithmus einfacherer sein, aber es werden immer alle Züge erstellt. Bei Spielen wie *Go*, bei dem die Zugerstellung relativ einfach ist, wird meist letztere Methode bevorzugt, wohingegen Spiele mit komplexen Zugberechnungen wie Schach, die inkrementelle Zugerstellung, Geschwindigkeitsvorteile bietet.

6.1 Selektive Zugerstellung

Im Jahr 1949 gliederte Claude Shannen die Schachalgorithmen in zwei Gruppen mit folgender Vorgehensweise:

- bestimme alle möglichen Züge, und alle daraus entstehenden möglichen Züge und dies rekursiv

- bestimme den ‚besten' Zug nach einem bestimmten Algorithmus, und darauf nur den besten gegnerischen Zug, und dies rekursiv

Ohne nachzudenken hört sich die zweite Möglichkeit sinnvoller an. Dies ist die Art und Weise Schach zu spielen. Allerdings zeigt die Praxis, dass Schachprogramme, die dieses implementieren (bisher ?) nicht erfolgreich sind.
Das Problem, das sich aus diesem Algorithmus ergibt besteht darin, dass der Computer quasi perfekt sein müsste bei der Auswahl des besten Zuges. Ein Beispiel: Man gehe davon aus, dass ein Programm 5 beste Züge verwendet, und dass sich mit einer Wahrscheinlichkeit von 95% (was wesentlich zu hoch gegriffen ist) der objektiv beste Zug darunter befindet. Bei einer Spiellänge von 40 Zügen ist die Wahrscheinlichkeit, dass er jeweils den besten Zug ermittelt hat weniger als 13 %. Selbst ein fast perfekter Algorithmus zur Zugerstellung mit einer Genauigkeit von 99 % würde statistisch gesehen mindestens einen Zug in jedem dritten Spiel nicht finden.
Mitte der 70ger wurde die komplizierte ‚Beste-Zug-Suche' verworfen. Die neu zu Verfügung stehende Rechenzeit, die durch den Wegfall der genauen Analyse des Schachbretts entstand, wurde dazu verwendet viel mehr Züge zu bewerten, was sich als erfolgreicher herausstellte.

6.1.1 Botwinnik

Ein extremes Beispiel der Vertreter der selektiven Zugerstellung ist Botwinnik, der Schüler von Schachweltmeister Mikhail Botwinnik war. Er war der Meinung, dass nur ein Schachprogramm, das die Denkweise eines Großmeisters besitzt, auch so gut spielen kann wie ein Großmeister. Er widmete sein Leben dem Computerschach und der selektiven Suche und scheiterte.

6.2 Komplette Zugerstellung

Die naheliegende alternative Möglichkeit zur ‚besten Zug Suche' ist die Erstellung aller möglichen Züge und einer anschließenden Bewertung nach Ausführung:

- erstelle alle möglichen Züge einer Spielbrettsituation
- sortiere sie auf eine Art und Weise, dass ein möglichst guter Zug als erster ausgeführt wird, um Geschwindigkeitsvorteile zu erreichen
- wende diese Züge an und bewerte sie, bis alle durchsucht sind. In bestimmten Fällen kann sie Suche & Bewertung bereits vorzeitig abgebrochen werden (näheres später)

Ältere Schachprogramme wie SARGON berechnen die möglichen Züge zu einem bestimmten Zeitpunkt zur Laufzeit, was viel Prozessorzeit benötigt. Heutzutage spielen diese wenigen Kilobyte, die die Datenbank zur Speicherung der möglichen Züge benötigt, keine Rolle mehr. Die vorberechneten Züge beschleunigen so die Suche. Mit Hilfe des *Transpositionsverzeichnisses* können bereits bewertete Züge überprüft werden, was ebenfalls die Suche beschleunigt.
Die Idee der kompletten Zugerstellung ist nicht nur einfach, sondern auch universell einsetzbar. In anderen Spielen (wie *Go*) können Züge nicht so einfach kategorisiert werden wie beim Schach (z.B.: ein Zug, der eine Figur schlägt, ...). Die folgende Technik wäre für diese Spiele ungeeignet:

6.3 Inkrementelle Zugerstellung

Anspruchsvolle Schauprogramme beschränken sich daher auf eine Erstellung von nur wenigen Zügen zu einem Zeitpunkt, die dann angewendet und bewertet werden, um bereits hier den Suchbaum beschneiden zu können. Dadurch kann es vorkommen, dass keine weiteren Zugmöglichkeiten erstellt werden müssen.
Folgende Fakten führten zum Erfolg dieser Art der Zugerstellung:

- in den 70gern bedeutete eine vorberechnete Zugdatenbank mit seinem nötigen Arbeitsspeicher eine kleineres *Transpositionsverzeichniss*. Somit wurde die Zugdatenbank verworfen und nur möglichst wenig Zugmöglichkeiten während der Laufzeit erstellt
- meistens ist ein Zug, der den Suchbaum beschneidet, ein Zug, bei dem eine Spielfigur geschlagen wird. Steht die Dame von Spieler A beispielsweise ungedeckt auf dem Spielbrett wird Spieler B erst versuchen diese Dame zu schlagen. Da es nur wenige Möglichkeiten für einen solchen Zug gibt, und die Bestimmung der Züge, die eine Figur schlagen relativ einfach zu bestimmen ist, wird der gesuchte Abbruch die Suche mit wenig Aufwand erreicht.

Deshalb berechnen viele Schachprogramme zuerst die Züge, bei denen andere Figuren geschlagen werden, um einen schnellen Abbruch der Suche zu erreichen. Es wurden einige Wege gefunden, diese Suche nach Zügen, die andere Figuren schlagen, zu beschleunigen. Eine mögliche Lösung verwendet wieder die *BitBoards*:
CHESS 4.5 verwendet dazu 2 x 64 *BitBoards*. Jedes der ersteren 64 *BitBoards* enthält alle Felder, die die Figur von diesem Feld aus angreift. Die zweiten enthalten alle Felder von Figuren, die dieses Feld des *BitBoards* angreifen. Sucht das Programm nach den Zügen, die die schwarze Dame schlagen würde, wird das *BitBoard* der Dame genommen und damit das *BitBoard* für den Angriff dieses Feldes indiziert. Nun werden nur Züge der Figuren erstellt, die das neue *BitBoard* ergeben.

6.4 Sortierung von Zügen

Die Effizienz des Suchalgorithmus ist stark davon abhängig, ob die Züge sinnvoll sortiert sind. Eine gute Sortierung der Züge hat einen Suchbaum zur Folge, der ungefähr die Größe der Quadratwurzel dessen hat, was ein schlechter Suchbaum mit schlecht sortierten Zügen ergeben würde.
Es ist offensichtlich, dass die Züge nach ihren Erfolgsaussichten sortiert sein sollten. Dazu müssen wieder Vermutungen getroffen werden, welches die bestmöglichen Züge sind: Man beginnt beispielsweise mit den Zügen, die andere Figuren schlagen, oder mit Zügen, die einen Bauern zum Offizier befördern (was die Materialbalance stark verändert), oder alle Züge, die den König ins Schach stellen (was meist nur sehr wenige sind). Danach folgen die Züge, die früher schon einen Abbruch im Suchbaum erzeugt haben, in der Hoffnung, dass sie es wieder tun (bezeichnet als ‚*Killer-Zug*'). Dies ist die Grundlage des schrittweise vertiefenden Alpha-Beta-Algorithmus. Diese Technik hat nichts mit selektiver Zugerstellung zu tun – es wird nicht nur der beste Zug verwendet. Im schlechtesten Fall müssen alle möglichen Züge berechnet werden, was bei der selektiven Variante nicht möglich wäre.

Noch ein Hinweis: im Schach gibt es einige Züge, die nicht erlaubt sind, da sie den König im Schach stehen lassen / stellen. Da dies recht selten der Fall ist und die Überprüfung auf diese Situation relativ viel Zeit kosten würde, wird diese Situation zuerst nicht betrachtet. Erst spä-

ter, wenn es wirklich ein guter Zug ist, wird auf diese Problematik hin geprüft, und der Zug eventuell verworfen.

7 Einfache Suchalgorithmen

Dieses Kapitel ist KI-Algorithmen von allgemeinen Strategiespielen gewidmet, die am Beispiel vom Schach erklärt werden. Im nächsten Kapitel werden diese Grundlagen speziell auf das Schachspiel optimiert.

7.1 Warum überhaupt suchen ?

Weil wir nicht clever genug sind, es ohne zu tun.

Ein wirklich intelligentes Programm könnte das Spielfeld hernehmen und daraus bestimmen, welche Seite im Vorteil wäre und welche Strategie gewählt werden müsste, um einen kleinen Vorteil zum Sieg zu führen. Aber hinsichtlich der Tatsache, dass es viele Muster zu untersuchen gibt, dass es viele Regeln und viele Ausnahmen zu diesen Regeln gibt, ist bisher noch kein Programm entwickelt worden, das dies verwirklicht hat – und es wird wahrscheinlich nie ein Programm geben, das es jemals umsetzten wird. Das Einzige was der Computer gut kann ist schnell rechnen. Deshalb versucht das Schachprogramm nicht von vornherein gute Züge zu finden, sondern sucht in die Tiefe, wobei er jeweils den besten Zug für seinen Gegner annimmt, um eine gute Zugkombination für sich zu finden.
Eine tiefe Suche ist einfacherer zu implementieren als irgendwelche komplizierten Taktiken. Beispielsweise die doppelte Schlagmöglichkeit für Springer: ein Zug, bei dem der Springer sich auf einem Feld befindet, von dem aus er zwei verschiede Figuren gleichzeitig bedroht (beispielsweise eine Dame und einen Turm). Um dafür eine logische Repräsentation zu finden muss ein nicht zu unterschätzender Aufwand betrieben werden, vor allem wenn noch hinzukommt, ob der Springer selbst bedroht ist und in wie weit die gegnerischen Figuren gedeckt sind. Dahingegen deckt eine einfache 3 Halbzüge Suche die komplette Problematik ab: die Suche wird von sich aus den Zug entdecken und den Springer so platzieren, dass er beide Felder bedroht. Dabei werden die beiden Angriffsmöglichkeiten getestet und später das Spielfeld bewertet, bei dem sich die Materialbalance geändert hat. Dadurch zeigt sich, ob der Zug sinnvoll ist oder nicht. Das Programm wird keine Möglichkeit auslassen und falls da eine bessere Kombination mit 5 Zügen existiert, wie seltsam sie auch sein mag, die über den Verlust der eigenen Dame führt, wird der Computer sie doch entdecken wenn die Suchtiefe tief genug ist. Daraus lässt sich ableiten, dass eine tiefe Suche sehr komplizierte und trickreiche Zugkombinationen hervorbringen wird.

7.2 MiniMax

Die grundlegende Idee von allen Suchalgorithmen für Spiele mit 2 Spielern ist der *MiniMax*-Algorithmus. Vor gut 60 Jahren war Von Neumann einer der ersten Personen, die diesen Algorithmus beschrieben.
Zusammgefasst kann der *MiniMax* folgendermaßen definiert werden:

- man geht davon aus, dass das Spielfeld bewertet werden kann, so dass man weiß, ob Spieler A (Max) im Vorteil ist, oder der Gegner (Min), oder ob die beiden Positionen ungefähr unentschieden sind. Diese Bewertung wird durch eine numerische Zahl repräsentiert: eine positive Zahl bedeutet, dass Max im Vorteil ist, eine negative Zahl, dass Min im Vorteil ist, bei null ist keiner dem anderen voraus.
- die Aufgabe von Max ist es, eine möglichst positive Bewertung zu erzielen

- die Aufgabe von Min ist es, eine möglichst hohe negative Bewertung zu erzielen
- man geht davon aus, dass beide Spieler fehlerfrei spielen und immer den Zug ausführen, der für sie am besten ist (nach dem Max / Min Kriterium)

Diese Regeln werden anhand einem einfachen Spiel, bei dem jede Partei genau 2 mögliche Züge hat verdeutlicht: Die Bewertungsfunktion wird nur auf das resultierende Spielbrett ausgeführt, nachdem beide Parteien (Min & Max) einmal gezogen haben.

Zug von Max	Zug von Min	Bewertung
A	C	12
A	D	-2
B	C	5
B	D	6

Max muss davon ausgehen, dass Min fehlerlos spielt. Daher weiß er, dass wenn er Zug A spielt sein Gegner mit D entgegnen wird, was zu einer Bewertung von –2 führen würde (Min gewinnt). Wenn Max jedoch B spielt, kann Min mit seinem besten Zug (diesmal C) nur eine Bewertung von 5 erreichen, d.h. Max gewinnt. Nach dem *MiniMax*-Algorithmus wird nun Max Zug B auswählen, auch wenn er mit A eine bessere mögliche Spielsituation erreichen könnte, wenn Min falsch ziehen würde.

Das beim Beispiel oben nicht offensichtliche Problem des *MiniMax*-Algorithmus besteht darin, dass der Zugbaum exponentiell mit der Suchtiefe wächst. Dies wird maßgeblich bestimmt von:

- der Anzahl der Zügen, die jeder Spieler machen kann – bezeichnet mit B (*Branching-Faktor*: Verzweigungsfaktor)
- die Suchtiefe d (depth), die normalerweise N-Halbzüge umfasst, bei der N die Anzahl der Züge beider Parteien zusammen ist.

Beim Schach ist der *Branching-Faktor* im Mittelspiel meist um die 35; in *Othello*, um die 8. Eine Suchtiefe von 8 Halbzügen würde das Durchsuchen von 1,5 Millionen möglichen Knoten bedeuten, da die Komplexität von *MiniMax* $O(B^N)$ ist. Kommt nur ein einziger weiterer Halbzug hinzu, würde der Suchbaum auf 30 Millionen Knoten anwachsen – bei N = 10 ergeben sich 1,8 Billionen Knoten !

Deshalb müssen Strategien verwendet werde, die diesen Suchbaum beschneiden – auf Kosten der Genauigkeit.

7.3 AlphaBeta

Angenommen, man hat bereits im obigen Beispiel für *MiniMax* den B-Zug von Max berechnet. Daher weiß man, dass die Bewertung für Max schlechtestenfalls 6 sein kann.
Wenn man nun Zug A untersucht und feststellt, dass die Zugkombination A-D einen Wert von –2 ergibt, was wesentlich schlechter als der bereits gefundene B-Zug ist, braucht man A-C gar nicht weiter zu untersuchen, da man davon ausgeht, dass Min immer den besten Zug zieht und somit bereits Zug A bereits schlechter ist, als der B Zug, auch wenn die maximal zu erreichende Bewertung höher wäre, was aber nicht interessiert.

Das ist die grundlegende Idee des *AlphaBeta*-Algorithmus: wenn man bereits einen guten Zug entdeckt hat kann man schnell viele andere Züge verwerfen, die sich schlecht auswirken würden. Mit Hilfe der bereits beschrieben *Transpositionstabelle* lässt sich die Komplexität des reinen *MiniMax*-Algorithmus auf ca. die zweifache Quadratwurzel reduzieren. In Zahlen ausgedrückt reduzieren sich die oben beschrieben 1,5 Millionen Knoten auf nur 2500.

7.4 Züge für den AlphaBeta vorsortieren

Was kann man tun, um ein möglichst schnelles erreichen eines guten *MiniMax* Wertes zu erreichen, damit die weitere Suche frühzeitig abgebrochen werden kann ?
Die Ursache für einen guten *MiniMax*-Wert ist ein guter Zug. Deshalb ist es wichtig zuerst die Züge auszuprobieren, die mit großer Sicherheit zu einem maximalen *MiniMax*-Wert führen. Der schlechteste Fall wäre, wenn die Züge genau umgekehrt sortiert wären, da dabei der komplette Suchbaum durchwandert werden müsste, was den *AlphaBeta*-Suchalgorithmus auf die Komplexität des *MiniMax* zurückführen würde.
Deshalb ist es wichtig die möglichen Züge vorher sinnvoll zu sortieren. Es müssen Vermutungen angestellt werden, welcher Zug am besten sein könnte. Einige Sortierkriterien sind:

- man bewertet die Züge mit Hilfe der allgemeinen Bewertungsfunktion nachdem jeder Zug ausgeführt wurde (Suchtiefe 1). Je besser die Bewertungsfunktion, desto besser werden die Züge danach sortiert sein.
- versuche einen Zug zu finden, der bereits im *Transpositionsverzeichnis* Bewertet wurde und über den der Suchbaum beschnitten werden kann
- Gliedern der Züge in verschiedene Klassen: eine Zug bei dem die eigene Dame gleich geschlagen wird ist selten erfolgreich und sollte daher an hinterer Stelle ausgeführt werden
- ein Zug, der bereits bei derselben Suchposition im Suchbaum einen Abbruch der Suche erzeugt hat, wird dies mit einer gewissen Wahrscheinlichkeit wieder tun. *Dieser ‚Killer-Zug'* basiert auf der Tatsache, dass einige Züge belanglos sind: Wenn die eigene Dame nicht gedeckt ist und geschlagen werden kann ist es egal ob man den Bauer h2 ein oder zwei Felder nach vorne zieht, der Computer wird jedesmal den ‚Läufer schlägt Dame' Zug wählen und dadurch ein Abbruch im Zugbaum erzwingen. Deshalb sollte der ‚Läufer schlägt Dame' Zug immer zuerst ausprobiert werden.
- eine Erweiterung des *‚Killer-Zuges'*: wenn im Laufe des Spiels sich der Zug g2 nach e4 als gut erwiesen hat, dann ist es geschickt einen ähnlichen Zug erneut auszuführen, auch wenn es beispielsweise die Dame statt dem Läufer ist, die den Zug jetzt ausführen kann. Diese Heuristik kann in zwei (für jede Seite einen) einfachen 64x64 integer Feldern implementiert werden.

Bisher waren es alles offensichtlich logische Ideen, die zum Erfolg geführt haben. Eine weitere nicht so offensichtliche ist die Suche mit einer schrittweise erhöhten Suchtiefe.

7.5 AlphaBeta mit schrittweiser erhöhten Suchtiefe

Wenn man die Züge mit Suchtiefe 6 bestimmen will wäre es geschickt, die Bewertung der Züge mit Suchtiefe 5 bereits zu kennen, um die Züge sinnvoll sortieren zu können.

Diese Idee wird mit der schrittweisen Erhöhung der Suchtiefe verfolgt: man beginnt bei einer Suchtiefe von 2 Halbzügen und verwendet das Ergebnis, um die Züge bei der Suche nach 3 Halbzügen neu zu sortieren, bis die gewünschte Suchtiefe erreicht ist.
Ohne groß zu überlegen scheint dies ein großer Aufwand zu sein, der dem Ertrag nicht aufwiegt, das stimmt aber nicht:
Man betrachtet die Größe des Suchbaums in Abhängigkeit der Suchtiefe d und des Verzweigungs-Faktors B. Der Baum besitzt B Knoten bei Suchtiefe 1, B * B Knoten bei Suchtiefe 2, B * B * B bei 3, usw. Daraus ergibt sich eine Suche mit der Suchtiefe d-1, die um B Knoten kleiner ist als der Baum mit der vollen Suchtiefe d. Falls B groß ist (ca. 35 beim Mittelspiel) folgt daraus, dass der größte Teil der Berechnung im letzten Schritt des Algorithmus mit der maximalen Suchtiefe entsteht. Durch die doppelte Suche mit einer Suchtiefe von d-1 würde sich falls es keinen Vorteil bringen sollte, ein vergrößerter Zeitaufwand von nur 4% ergeben.
Da sich aber die bereits berechneten Positionen und Bewertungen wiederverwerten lassen, kann diese Idee die Performance des Schachprogramms wesentlich steigern. Durch die vorsortierten Züge erreicht man wesentlich mehr Abbrüche bei der Suche im Suchbaum. Somit werden beim *AlphaBeta* mit schrittweiser erhöhter Suchtiefe wesentlich weniger Knoten durchsucht, als wenn man den *AlphaBeta* direkt mit der maximalen Suchtiefe verwenden würde. Zudem sind viele der Bewertungen mit geringer Suchtiefe bereits im *Transpositionsverzeichnis* gespeichert und müssen gar nicht neu berechnet werden.

7.6 Die Spielweise des Computers

Mit Hilfe des AlphaBeta-Algorithmus mit schrittweiser erhöhter Suchtiefe und des *Transpositionsverzeichnisses* kann der Computer bei jeder Schachbrettstellung einige Züge im voraus berechnen und somit vernünftig spielen. Man sagt, dass MiniMax und seine Erweiterungen durch seine Eigenschaften Schach auf eine besondere Art und Weise spielen, die nicht der des Menschen entspricht und auch nicht sonderlich spektakulär ist.
Ein Beispiel dafür: man geht davon aus, dass der Computer eine Spielsituation bis in die Tiefe von 8 Halbzügen berechnet hat. Er entdeckt dabei einen Zug, bei dem er immer gewinnen wird und den Gegner Schachmatt setzten kann. Es gibt nur eine einzige Ausnahme, die ziemlich schwierig und trickreich ist – aber der Computer kennt sie zwangsweise. Bei einem Spiel zwischen Großmeister könnte so ein Zug in die Geschichtsbücher eingehen – der Computer hingegen wird ihn nicht verwenden, da er immer von der für ihn schlechtesten Möglichkeit ausgeht.
So passiert es, dass wenn der Computer eine Zugmöglichkeit entdeckt, die zwangsweise zu einem Remis führt, er ihn verwenden wird, falls er der Meinung ist, dass es besser ist unentschieden zu spielen, als etwas riskanter auf Sieg zu spielen, immer in Hinblick darauf, dass er denkt sein Gegenspieler sei perfekt.
Daraus entsteht die sehr vorsichtige Spielweise von Computern, die immer denken einen Weltmeister als Gegner zu haben. Kennt man diese Schwäche und die Tatsache, dass Computer nicht unendlich tief suchen können, können gezielt Züge angewendet werden, die der Maschine viel Zeit kosten, um einen guten Gegenzug zu finden. Zudem kann mit einer Kombination, die größer ist als die Suchtiefe des Computers (evtl. 10 Züge), der Computer in eine Falle gelockt werden. Es ist wichtig, dass man vorher seinen Gegner studiert, wie es auch im normalen Schach der Fall ist. Hätte Kasparov die Möglichkeit gehabt länger gegen *DEEP BLUE* zu trainieren hätte er mit großer Sicherheit dieses Duell gewonnen, da bisher alle Programme Schwächen aufweisen.

8 Weiterentwickelte Suchalgorithmen

Die bisher vorgestellten Suchalgorithmen lassen sich weiter auf die Schachproblematik optimieren, um eine höhere Suchtiefe zu erlangen.
Bisher haben die Suchalgorithmen bis zu einer fest einprogrammierten Suchtiefe gerechnet. Wenn man davon ausgeht, dass ein Schachprogramm nur den Algorithmus des *AlphaBeta* mit schrittweise erhöhter Suchtiefe bis Tiefe 5 implementiert hat, kann folgendes passieren:

- in einer bestimmten Spielsituation entdeckt der Computer einen Zug, mit dem er den Gegenspieler innerhalb 3 Zügen matt setzten kann. Es ist Schwachsinn die Suche bis zum 5. Zug weiterzuführen, was zu unbestimmten Zuständen im Programm führen könnte
- eine andere Situation: es gibt einen Zug, bei dem es möglich ist im 5. Zug einen gegnerischen Bauern zu schlagen. Der Computer betrachtet diesen Zug als erfolgreich, da er die Materialverteilung zu seinen Gunsten lenkt und führt ihn aus. Hätte der Computer nur einen Halbzug tiefer gerechnet hätte er entdeckt, dass genau dieser Zug die eigene Dame schutzlos freigibt und sie damit geschlagen werden kann
- ein dritter Fall: angenommen, der Dame des Computer wurde eine Falle gestellt. Der menschliche Spieler wird im 4. Halbzug die Dame schlagen, außer der Computer führt einen Zug aus, bei dem er seinen Läufer opfert. Dabei entdeckt er nicht, dass dieser Zug das Schlagen der Dame nur auf den 6. Halbzug verzögert, da er nur bis Zugtiefe 5 rechnet. Er würde also sinnlos seinen Läufer hergeben. Einen Zug später entdeckt er dies, was aber nichts mehr hilft. Dieses Problem wurde von Hans Berliner als „*Horizonteffekt*“ beschrieben. Er entwickelte sogleich eine Lösung dafür: die „*Ruhesuche*“ (*quiescence search*).

Zusammengefasst ist das Problem der oben beschriebenen Beispiele folgendes: die normalen Schachsituationen sind zu komplex, um eine faire Bewertungsfunktion auf sie anwenden zu können. Das Spielfeld kann nur in ‚ruhigen‘ Situationen bewertet werden.

8.1 Ruhesuche

Es gibt zwei Möglichkeiten eine Spielsituation abzuschätzen: dynamische Entwicklung (mit dem Hintergrund, was sich aus dieser Stellung entwickeln kann) und statische Bewertung (betrachtet nur die aktuelle Spielsituation – wie Material, Bauernformation, ... – ohne die Konsequenzen zu betrachten, die sich daraus ergeben). Die dynamische Entwicklung wird mit der Suche erreicht, die statische Bewertung darf, wie bereits erwähnt, nur angewendet werden, wenn keine folgenträchtigen Züge direkt darauf folgen können. Diese Stellungen werden als „ruhig“ oder „ruhend“ bezeichnet und über die *Ruhesuche* gefunden.
Die Grundlage der *Ruhesuche* ist folgende: wenn das Programm beispielsweise die Suche bis Suchtiefe 6 durchgeführt hat wird weiter gesucht bis es nur noch Züge gibt, die solche Ruhezüge sind. Auf diese Situation wird die Bewertungsfunktion angewendet.

Um solche ruhigen Stellungen entdecken zu können muss man wissen, welche Spielzüge die Situation drastisch verändern können. Beim Schach spielt sicherlich die Materialverteilung eine große Rolle: das Schlagen von Figuren (vor allem von Offizieren) sowie die Bauernverwandlung. Dazu kommen noch Möglichkeiten, den Gegner in Schach zu stellen, vor allem wenn dadurch der Gegner Schachmatt gesetzt werden kann. Betrachtet man das ganze bei *Go*

hat man das Problem, dass sich keine klassischen Ruhesituationen finden lassen, da sich die Punkteverteilung mit jedem Zug drastisch ändern kann.
Implementiert man die Ruhesuche als eine, die den normalen Suchalgorithmus so erweitert, dass sich nach der normalen Suchtiefe die Spielfiguren solange schlagen, bis kein gültiger Schlagzug mehr vorhanden ist, ergibt dies weitere Verzweigungen von ungefähr 4-6 Schlagzügen pro Ausgangsstellung, die schnell zu 0 konvergieren. Da die *Ruhesuche* aber sehr oft aufgerufen werden muss (vor jeder Bewertung), verbringt das Schachprogramm bis zu 50% seiner Zeit in dieser Routine.
Mit Hilfe einer solchen Ruhesuche kann der Problematik des *Horizonteffekts* wirkungsvoll begegnet werden.

8.2 Null-Zug

Eine sehr effiziente Methode, zur Beschleunigung eines Schachprogramms ist die Verwendung von *Null-Zügen*.
Der *Null-Zug* besagt, dass der Gegner zweimal hintereinander ziehen darf – der eigene Zug wird zu Null. In den meisten Situationen wäre eine *Null-Zug* sinnlos, da er immer Vorteile für den Gegner bringt. Falls dieser seine doppelte Zugmöglichkeit nicht dazu nutzen kann, seine Situation wesentlich zu verbessern scheint das Spielfeld in Ruhe zu sein.
Wenn der Computer den *Null-Zug* bei der Suche verwendet ergeben sich folgende Vorteile hinsichtlich Geschwindigkeit und Genauigkeit:

- angenommen, die Situation ist sehr eindeutig zum Vorteil des menschlichen Spielers. Selbst wenn er einen Zug auslässt kann der Gegenspieler keinen Zug finden, der seine Situation wesentlich verbessern könnte. Falls dieser Zug beispielsweise bei einer Suche mit Tiefe N angewendet wird, ergibt dies nur noch ein Suchbaum für den *Null-Zug* mit der Tiefe N-1. B sei 35 – wie im typischen Mittelspiel beim Schach. Daraus ergibt sich ein Extraaufwand von 3% für die *Null-Zug*-Berechung, falls er keine Vorteile bringt. Wenn er hingegen einen Abbruch im Suchbaum erzeugt, hat man 97% des Suchaufwandes eingespart, was sehr oft passiert !
- in einem anderen Fall kann der *Null-Zug* für die Ruhesuche genutzt werden. Angenommen, die Spielfeldstellung erlaubt nur einen einzigen Zug, der eine Figur schlägt: der eigene Turm schlägt einen Bauern, danach wird er sofort vom gegnerischen Springer geschlagen, der nicht wieder geschlagen werden kann. In diesem Fall wäre es sinnvoller, keine Figur zu schlagen. Hier kommt der *Null-Zug* ins Spiel. Dieser wird bei der *Ruhesuche* eingestreut, und falls er erfolgreicher ist, als ein Zug bei dem eine Figur geschlagen wird, ist es eine gute Situation das Spielfeld zu bewerten.

Über den Trick mit dem *Null-Zug* kann der Suchbaum nochmals um 25% - 70% beschnitten werden. Dabei ist es sehr einfach einen *Null-Zug* zu implementieren – es muss einfach nur der aktive Spieler getauscht werden.

8.3 Aspirierte Suche und MTD(f)

Dem normalen *AlphaBeta* ist es egal, welchen Wert der *MiniMax*-Wert annimmt. Er untersucht alles (wenn er keinen Abbruch in der Suche erzielt), auch wenn der Wert noch so sinnlos ist. Wenn man aber davon ausgeht, dass man den ungefähren Zielwert des *MiniMax* vorher bestimmen kann (beispielsweise durch die schrittweise Erhöhung der Suchtiefe und deren

Ergebnisse), kann man bereits beim Durchsuchen des Suchbaumes Situationen erkennen, die nicht zum Ziel führen würden und man wird die weitere Untersuchung abbrechen.
Angenommen der Erwartungswert für die Situation ist 0, da das Spiel sehr ausgeglichen ist. Wird nun ein Suchpfad untersucht, bei dem die Bewertung bereits +20.000 ergibt, kann dieser mit ziemlich großer Sicherheit verworfen werden.
Das ist die Idee der *Aspirierten Suche* – eine Variante das *AlphaBeta*, bei der die Grenzen (der Wert von *MiniMax* nach der Bewertung) nicht von - ∞ bis + ∞ gehen, sondern ein kleines fest vorgegebenes Fenster darstellen. Wenn die Suche erfolgreich ist wird viel Zeit gespart, da die anderen Suchpfade bereits früh abgebrochen werden konnten. Wenn kein gültiger Zug für dieses Fenster gefunden wird erweitertet man das Fenster. Dadurch entstehen natürlich Mehrkosten auf Grund der erneuten Suche mit dem erweiterten Fenster. Diese Optimierung steht und fällt mit dem Wert, der für den Zug vorhergesagt wird.
Mitte der 90ger Jahre entwickelte Aske Plaat die Idee der *Aspirieten Suche* weiter: was passiert, wenn man die Fensterbreite gleich 0 setzt ? Der Algorithmus würde scheitern – aber der Suchbaum wäre auch sofort durchlaufen. Wenn durch den Suchabbruch ersichtlich wird, dass der Wert zu hoch gewählt wurde wird eine erneute Suche mit einer geringeren Zielwertung aufgerufen.
Diese geniale Idee hat sich im *MTD(f)* Algorithmus niedergeschlagen, der nur ca. 10 Zeilen lang und sehr effizient ist. Zudem ist er gut geeignet für grobe Bewertungsfunktionen. Das kann man leicht daran erkennen, dass ein Fenster der Größe 0 länger dafür brauchen wird die Suche mit einer kleinsten Einheit von 0.001 Bauern durchzuführen, als eine mit 0.01 Bauern.
Es gibt noch weitere *AlphaBeta*-Algorithmen, wie beispielsweise den *NegaScout*, doch nach Plaat ist *MTD(f)* zurzeit der effizienteste Algorithmus.

8.4 Vereinzelte Ausdehnung (Singular Extension)

Beim Schachspielen ist es offensichtlich, dass es einige wenige Züge gibt, die sinnvoll sind und sehr viele die zu keinem guten Ende führen. Man sollte daher nicht zu viel Zeit verschwenden um nach alternativen guten Zügen Ausschau zu halten.

Angenommen, man verwendet den *AlphaBeta* mit schrittweiser erhöhter Suchtiefe und ist bereits bei Suchtiefe N-1 angelangt. Es wurden nur Züge entdeckt, die eine negative Bewertung zur Folge haben, bis auf einen, der die gegnerische Dame schlägt und damit sehr positiv scheint. Wenn es wichtig ist, Zeit einzusparen (z.B. unter Turnierbedingungen), kann die Suche mit Tiefe von N auf eine Suche auf N-1 reduziert werden, bei der man nur die besten Züge weiter verfolgt. Durch diesen Trick kann, wie bereits oben erwähnt, bei einem Verzweigungsfaktor von 35, 35 97% der benötigten Rechenschritte eingespart werden.
Die Entwickler von *DEEP BLUE* haben diese Idee weiter verfolgt und damit die Vereinzelte Ausdehnung von Zügen hervorgebracht. Wenn bei der Suche ein sehr guter Zug auftritt wird dieser mit einer höheren Suchtiefe weiter verfolgt, um eventuelle Fallen zu erkennen. Dabei ist es sehr wichtig nur sehr ausgewählte Züge tiefer zu verfolgen, da sonst schnell unverhältnismäßig viel Zeit darauf verwendet wird, durch den exponentiell wachsenden Suchbaum.

9 Bewertungsfunktionen

Die oben angeführten Suchalgorithmen und Datenstrukturen verwenden als Auswahlkriterium meist eine Bewertungsfunktion. Deswegen ist es wichtig, sich einen ausgeklügelten Algorithmus dafür zu überlegen. Es ist bei den meisten Spielen unmöglich eine objektive Bewertung der aktuellen Spielsituation zu geben. Obwohl die Regeln des Schachs bekannt sind und die aktuelle Spielbrettsituation einsehbar ist werden bereits einfache Bewertungsstrategien sehr komplex. Hier einige Ideen, ein Spielbrett zu bewerten:

9.1 Materialverteilung

Vereinfacht gesagt ist die Materialverteilung die Summe der einzelnen Spielfiguren auf dem Brett. Ein Blick in die Schachliteratur verrät folgendes, übliches Verhältnis: Dame 900, Turm 500, Läufer 325, Springer 300 und Bauer 100; der König ist unendlich viel wert. Allein mit diesem Wissen kann man ein ansatzweise intelligentes Schachprogramm verwirklichen.
Ab und zu kann es sinnvoll sein eine eigene Figur zu opfern (selbst eine Dame). Diese Variante wird durch die Suchtiefe beim Suchalgorithmus abgedeckt. Wenn das Opfern der Dame zum Schachmatt des Gegners führt wird der Suchalgorithmus ihn entdecken, ohne dass man spezielle Abfragen implementieren muss.
Nur sehr wenige Programme verwenden diese primitive Version der Bewertung. Da die Berechnung sehr einfach ist, werden zusätzliche Gesichtspunkte eingebaut. Zum Beispiel wenn man eine Übermacht an Figuren besitzt, ein Austausch zweier gleichwertiger Figuren sinnvoll ist. Oder die Tatsache, dass ein Bauernopfer, das im Eröffnungsbuch steht, sinnvoll sein kann. Dies kann mit Hilfe eines Ausgleichsfaktors eingebaut werden, der bewirkt, dass die Bewertungsfunktion weiß, dass sie im Vorteil ist obwohl die reine Materialbilanz -150 Punkte ergeben würde.

9.2 Bewegungsfreiheit und bedrohte Felder

Eine Bedingung, die erfüllt sein muss, um einen Gegner Schachmatt zu setzten ist die Tatsache, dass sich der König nicht mehr bewegen kann. Daraus lässt sich einfach schlussfolgern, dass eine gewisse Mobilität von Vorteil ist: ein Spieler kann meist eher bei 30 möglichen Zügen einen guten entdecken, als bei 3.
Beim Schach kann die Mobilität einfach bestimmt werden, indem man alle möglichen Züge für seine eigenen Spielfiguren zählt. Doch zeigt sich, dass eine solche Bewertung nicht all zu viel Sinn macht: wenn man den Bauern vor dem Turm betrachtet würde dies bedeuten, dass es sinnvoll wäre ihn immer weiter nach vorne zu ziehen, um einen höheren Bewegungsfreiraum für den Turm zu erreichen. Andererseits wäre es wichtig, die gegnerische Bewegungsfreiheit durch zwanghaftes Schachstellen des gegnerischen Königs einzuschränken, was dazu führen würde, dass die eigenen Figuren über das ganze Spielbrett verteilt würden, was ebenso sinnlos wäre.
Trotzdem gibt es einige Grundregeln, die implementiert werden können:

- Springer am Rand ist eine Schand
- ein eingesperrter Läufer hat oft weniger Nutzen
- der eigene Bauer direkt vor dem Turm in der Ausgangsstellung schränkt die Zugmöglichkeit stark ein

Bei der Bewertung der Felder, die bedroht werden, zeigen sich Parallelen zur Bewertung der Bewegungsfreiheit. Beim Schach besitzt ein Spieler ein Feld wenn mehr eigene Figuren das Feld bedrohen wie gegnerische (vereinfacht gesagt – natürlich ist es nicht sinnvoll die Dame auf ein Feld zu ziehen, die von einem Bauern gedeckt wird). Vor allem die Mitte des Spielbretts steht beim Eröffnungsspiel im Mittelpunkt. Es stellt sich als schwierig heraus, eine Bewertung für die bedrohten Felder zu erstellen. Aber trotzdem haben viele Schachprogramme einen Algorithmus dafür eingebaut.

9.3 Entwicklung

Eine der bekannten Weisheiten fürs Schach ist die Tatsache, dass Offiziere wie Läufer und Springer möglichst bald ins Spiel gebracht werden sollten, dass es sinnvoll ist eine frühe Rochade auszuführen und dass Turm und Dame später für gezielte Angriffe genutzt werden sollten. Dafür gibt es viele Gründe: Läufer und Springen (sowie Bauern) kontrollieren das Mittelfeld, können Angriffe der Dame unterstützen und erlauben den Türmen in der hinteren Reihe größere Bewegungsfreiheit, falls diese sich entwickelt haben. Später im Spiel kann ein Turm, der die 7. Reihe der gegnerischen Bauern aufräumt, verheerende Folgen haben.
Um diese Ideen in die Bewertungsfunktion einfließen zu lassen können beispielsweise Springer oder Läufer, die sich in der hinteren Reihe befinden mit einem negativen Bewertungsfaktor belegt werden. Genauso verhält es sich mit einer Rochade. Verliert ein König bei einem Zug sein Rochaderecht ohne zu rochieren wird dies negativ bewertet....
Die Entwicklung der eigenen Figuren spielt in der Anfangsphase des Spiels eine entscheidende Rolle. Später im Spielverlauf (ab beispielsweise Zug 10), können einige der hier beschriebenen Regeln verworfen werden.

9.4 Bauernstruktur

Die Bauern sind die Seele einer Schachpartie – hört man immer wieder von Großmeistern des Schachs. Diese Aussage ist für einen Schachanfänger nicht unbedingt nachvollziehbar, doch ist es bemerkenswert wie oft bei einer guten Partie ein Vorteil von nur einem Bauer das Spiel entscheiden kann. Die Schachliteratur weist dabei auf einige Besonderheiten hin:

- Doppelt- oder Dreifachbauern (2 oder mehr Bauern in einer selben vertikalen Reihe): diese Art von Bauern behindern sich nur gegenseitig in ihren Zugmöglichkeiten
- vorbeigezogene Bauern: Bauern, die an den gegnerischen Bauern vorbeigezogen sind (die sie schlagen oder blockieren können) sind sehr mächtig, da die Bauernverwandlung nicht weit ist
- isolierte Bauern: ein Bauer, der keine eigenen Nachbarbauern mehr besitzt sollte anderweitig gedeckt werden.
- acht Bauern: zu viele Bauern auf dem Spielbrett behindern die eigenen Zugmöglichkeiten

Es gibt noch weitere Sonderregeln: Ein vorbeigezogener Bauer in einer Reihe mit einem Turm ist sehr mächtig, da die Figur, die den Bauern schlägt sofort vom Turm geschlagen werden könnte.

9.5 Bedrohung des Königs

Wie bereits oben erwähnt ist es nicht sehr wichtig, den König bei der Eröffnung oder im Mittelspiel zu schützen. Dies kann am einfachsten über eine Rochade erreicht werden.
Im Endspiel, wenn die meisten Figuren vom Brett sind, wird der König zu einer starken Figur. Es wäre Ressourcenverschwendung ihn hinter einer Reihe von Bauern zurückzulassen.
Die Bedrohung für einen König kann unter anderem daran gemessen werden, wie nahe sich gegnerische Figuren um eigenen König befinden. Je mehr Figuren und je näher sie sind, desto gefährlicher ist es. Dies ist natürlich auch von der Spielfigur selbst abhängig, aber die reine Entfernung liefert eine gute Abschätzung.

9.6 Die richtige Gewichtung

Wenn die Kriterien, nachdem die Bewertungsstrategie erfolgen soll, feststehen, muss eine sinnvolle Gewichtung der verschiedenen Faktoren bestimmt werden.
Dabei gerät man schnell in die Situation, dies und jenes gegenseitig Aufzuwiegen und an Feinheiten herumzufeilen. Dabei sollte man folgende Gesichtspunkt bedenken: es ist meist schwieriger, das Programm über eine Bewertungsfunktion stark zu machen, als einfach einen Halbzug tiefer zu suchen – deshalb sind in der Regel nicht zu komplexe Bewertungsfunktionen die besseren.

10 Quellen

Chess Programming
http://www.gamedev.net/reference/programming/features/chess1/

Computerschach - Geschichte
http://www.computerschach.de/einleit/start.htm

DEEP FRITZ gegen Kramnik
http://www.chessica.de/deepfritz.html

DEEP BLUE gegen Kasparov
http://www.computerschach.de/einleit/kapitel5.htm#blue

GNU Chess
http://www.gnu.org/software/chess/chess.html

Lehrbuch des Schachspielens
29. Auflage – 1996
Dufresne Mieses, Reclam

NegaScout
http://www.trinimon.de/Chess/ChessGame/Develop/chs_neg.htm

MTD(f) – Algorithmus
http://theory.lcs.mit.edu/~plaat/mtdf.html

Small Potatoe – Schachprogramm
http://smallpotato.sourceforge.net

Spielregeln
http://schachclub-baden-oos.haberichter.net/fideregeln.htm

WinBoard
http://www.tim-mann.org/xboard.html

WinBoard Communikationsprotokoll
http://www.tim-mann.org/xboard/engine-intf.html

11 Glossar

AlphaBeta	Verbesserter MiniMax-Algorithmus. Alpha entspricht Min, Beta entspricht Max. (Kapitel 7.3)
Aspirieten Suche	Suchalgorithmus, der auf ein MiniMax-Problem angewendet werden kann (Kapitel 8.3)
Bitboard	Ein Bitboard ist 64 Bits groß, bei dem jedes Bit ein Spielfeld repräsentiert. (Kapitel 5.1)
Branching-Faktor	Die Anzahl der im Schnitt möglichen Züge in einer Spielsituation und damit ein maßgeblicher Faktor zur Bestimmung der Suchknoten (Kapitel 4.3)
Deep Blue	Das bekannteste Computerschachprogramm überhaupt. Weitere Infos siehe Quellen
Deep Fritz	Das Computerschachprogramm, das zuletzt 2002 gegen einen Weltmeister antrat. Weitere Infos siehe Quellen.
doppeltes Hashing	Sichert einen Hashwert ab. Siehe Hashlock
en passant	Schlagen im Vorrübergehen für Bauern. Siehe Quellen: Spielregeln
Fritz	Das bekannteste Schachprogarmm.
Go	Altes Spiel aus dem China. Weitere Infos unter http://www.dgob.de/
Hashwert	Siehe Hashschlüssel
Hashlock	Ein 2. Hashschlüssel, der auf die selbe Situation einen anderen Hashalgorithmus anwendet, um eine Sicherheit für den Hashschlüssel zu gewinnen
Hashschlüssel	Berechnet über einen Algorithmus eine Zahl aus einer gegebenen Situation (beispielsweise ein Schachbrett)
Hashtabelle	Tabelle, auf die über Hashschlüssel zugegriffen wird
Horizonteffekt	Computer kann die Konsequenzen aus einer Boardstellung nicht absehen (Kapitel 4.3)
Killer-Zug	Ein Zug, der bei einer vorherigen Suche ein Abbruch im Suchalgorithmus erzeugt hat und dies mit einer gewissen Wahrscheinlichkeit wieder tut
MiniMax	Grundlage eines jeden Schachalgorithmus (Kapitel 7.2)
MTD(f)	Ein aspirierter Suchalgorithmus (Kapitel 8.3)
NegaScout	Ebenfalls ein aspirierter Suchalgorithmus siehe Quellen und (Kapitel 8.3)
Null-Zug	Ein Zug, in dem eine Partei nicht zieht. (Kapitel 8.2)
Offiziere	Offiziere sind beim Schach diejenigen Spielfiguren, die keine Bauern sind und somit in der hinteren Reihe starten
Othello	Ebenfalls ein kompexes Spiel: siehe http://www.othello-club.de.vu/
quiescence search	Siehe Ruhesuche
Rochade	Sonderzugregel des Königs. Siehe Quellen: Spielregeln
Ruhesuche	Eine Möglichkeit dem Horizonteffekt zu begegnen (Kapitel 8.1)
singular extensions	Durchsuchen einzelner weniger Zugkombination in die Tiefe (Kaptiel 8.4)
Transpositionsverzeichnis	Verzeichnis, in dem bereits bewertete Spielstellung gecached werden (Kapitel 5.2)

12 Anhang

12.1 Profilerlauf

```
Profile: Function timing, sorted by time
Date:    Tue Jul 08 08:21:39 2003


Programmstatistik
-----------------
    2003 Jul 07 16:36 Befehlszeile: "c:\chess\Release\BaChess"
    Gesamtzeit: 4208549,359 Millisekunden
    Zeit ausserhalb von Funktionen: 0,802 Millisekunden
    Aufruftiefe: 39
    Gesamtfunktionen: 1189
    Funktionsabdeckung: 43,7%
    Berechneter Verwaltungsaufwand: 4
    Durchschnittlicher Aufwand: 4

Modulstatistik fuer bachess.exe
-------------------------------
    Zeit innerhalb des Moduls: 4208548,557 Millisekunden
    100,0% Prozent im Modul
    1189 Funktionen im Modul
    Modulfunktionsabdeckung: 43,7%

        Funkt.        Funkt.+Unterfkt.      Anzahl
        Zeit     %         Zeit       %          Aufrufe  Funktion
-----------------------------------------------------------------------
  828587,674  19,7   828587,674  19,7 102315836 Board::FindWhitePiece(int) (board.obj)
  728812,076  17,3  1525319,775  36,2 21649384 BoardEvaluator::EvalKingTropism(class Board *,int) (boardevaluator.obj)
  440879,240  10,5   440879,240  10,5 50211059 Board::FindBlackPiece(int) (board.obj)
  372332,840   8,8   755243,353  17,9 21649384 BoardEvaluator::AnalyzePawnStructure(class Board *,int) (boardevaluator.obj)
  150235,781   3,6   150235,781   3,6 777893473 Move::~Move(void) (move.obj)
  146118,017   3,5   146118,017   3,5 701354590 Move::Move(void) (move.obj)
  118108,376   2,8   118108,376   2,8 584206659 Board::AddPiece(int,int) (board.obj)
  107914,702   2,6   226020,288   5,4 22468728 Board::StartingBoard(void) (board.obj)
   99856,658   2,4  3908655,272  92,9 21817876 AISearchAgentNegaScout::QuiescentSearch(class Board *,int,int,int)
   72372,853   1,7   149066,599   3,5 21649384 BoardEvaluator::EvalDevelopment(class Board *,int) (boardevaluator.obj)
   66698,442   1,6    66698,442   1,6 31337714 Board::HashLock(void) (board.obj)
   60278,780   1,4   121589,914   2,9 20591843 MoveListGenerator::ComputeWhiteQuiescenceBishopMoves(class Board *,int)
   55260,097   1,3   112371,368   2,7 21448396 MoveListGenerator::ComputeBlackQuiescenceBishopMoves(class Board *,int)
   49422,029   1,2    98878,616   2,3 20572271 MoveListGenerator::ComputeWhiteQuiescenceRookMoves(class Board *,int)
   48789,796   1,2    48789,796   1,2 22927493 Board::HashKey(void) (board.obj)
   48445,942   1,2    97223,883   2,3 21936855 MoveListGenerator::ComputeBlackQuiescenceRookMoves(class Board *,int)
   36864,344   0,9    36864,344   0,9 103319631 operator delete(void *) (delop.obj)
   36232,815   0,9  2509203,598  59,6 21649384 BoardEvaluator::EvaluateComplete(class Board *,int,bool) (boardevaluator.obj)
   35993,417   0,9    35993,417   0,9 174674417 Move::operator<(class Move const &) (move.obj)
   35984,880   0,9   691066,912  16,4 21649339 MoveListGenerator::ComputeQuiescenceMoves(class Board *)
   33349,635   0,8    68145,402   1,6 10212752 MoveListGenerator::ComputeBlackQuiescenceKnightMoves(class Board *)
   33151,774   0,8    72356,917   1,7 10098752 MoveListGenerator::ComputeWhiteQuiescenceKnightMoves(class Board *)
   28207,608   0,7    28207,608   0,7 138452221 Move::`scalar deleting destructor'(unsigned int) (move.obj)
   28207,608   0,7    28207,608   0,7 138452221 Move::`vector deleting destructor'(unsigned int) (move.obj)
   27849,934   0,7    27849,934   0,7 127942479 std::_Construct(class Move *,class Move const &) (movelistgenerator.obj)
   26629,817   0,6    43604,117   1,0 74046748 std::vector<class Move,class std::allocator<class Move> >::_Ufill(class Move
   25428,918   0,6    54603,946   1,3 22468705 MoveListGenerator::~MoveListGenerator(void) (movelistgenerator.obj)
   25160,805   0,6    91233,576   2,2 22927471 TranspositionTable::LookupBoard(class Board *,class TranspositionEntry * *)
   24455,693   0,6    84917,058   2,0 14813364 std::_Insertion_sort_1(class std::reverse_iterator<class Move *,class
   22577,433   0,5    22577,433   0,5 45855833 std::basic_string<char,struct std::char_traits<char>,class
   22223,041   0,5    41129,484   1,0 32771278 std::_Unguarded_insert(class std::reverse_iterator<class Move *,class
   21359,829   0,5    36715,052   0,9 45854732 std::basic_string<char,struct std::char_traits<char>,class
   18089,376   0,4    20296,210   0,5 74934214 std::vector<class Move,class std::allocator<class Move> >::_Ucopy(class Move
   17036,368   0,4    39613,765   0,9 45855778 std::basic_string<char,struct std::char_traits<char>,class
   17006,926   0,4    21959,246   0,5 22468726 MoveListGenerator::MoveListGenerator(void) (movelistgenerator.obj)
   16066,664   0,4    16066,664   0,4 72669706 std::fill(class Move *,class Move *,class Move const &)
   15549,140   0,4    29462,606   0,7 10190339 MoveListGenerator::ComputeWhiteQuiescenceKingMoves(class Board *)
   14515,289   0,3    26259,320   0,6 10720286 MoveListGenerator::ComputeBlackQuiescenceKingMoves(class Board *)
   12411,708   0,3   158556,819   3,8 10662437 std::_Stable_sort(class std::reverse_iterator<class Move *,class Move,class
   11480,058   0,3    11480,058   0,3 21649383 BoardEvaluator::EvalPawnStructure(int) (boardevaluator.obj)
   10932,253   0,3    19759,398   0,5 15374496 std::vector<class Move,class std::allocator<class Move> >::insert(class Move
    9879,814   0,2    14078,916   0,3 23845993 std::vector<class Move,class std::allocator<class Move> >::_Destroy(class Move
    9645,953   0,2     9645,953   0,2 45855041 std::basic_string<char,struct std::char_traits<char>,class
    9645,953   0,2     9645,953   0,2 45855041 std::vector<class std::vector<int,class std::allocator<int> >,class
    9645,953   0,2     9645,953   0,2 45855041 std::vector<int,class std::allocator<int> >::`default constructor
    9266,141   0,2     9266,141   0,2 21649384 BoardEvaluator::EvalRookBonus(class Board *,int) (boardevaluator.obj)
    8642,143   0,2    12070,993   0,3  9967587 MoveListGenerator::ComputeWhiteQuiescencePawnMoves(class Board *)
    8345,229   0,2     8345,229   0,2 21649384 BoardEvaluator::EvalBadBishops(class Board *,int) (boardevaluator.obj)
    8339,564   0,2  4154884,337  98,7  1109510 AISearchAgentNegaScout::NegaScout(class Board *,int,class Move *,int,int)
    8253,232   0,2    17524,374   0,4   729186 std::_Buffered_merge(class std::reverse_iterator<class Move *,class Move,class
    8193,072   0,2   234213,348   5,6 22468727 Board::Board(void) (board.obj)
    8031,210   0,2     8125,379   0,2 22926723 Board::ApplyMove(class Move *) (board.obj)
    7994,305   0,2     7994,305   0,2 21649384 BoardEvaluator::EvalBadKnights(class Board *,int) (boardevaluator.obj)
    7857,270   0,2    11116,858   0,3 10054950 MoveListGenerator::ComputeBlackQuiescencePawnMoves(class Board *)
    7070,841   0,2     7070,841   0,2 10662437 operator new(unsigned int,struct std::nothrow_t const &) (newop2.obj)
    6990,331   0,2    14505,413   0,3  1814926 std::merge(class std::reverse_iterator<class Move *,class Move,class Move
    6844,046   0,2     6844,046   0,2 22926704 Board::Clone(class Board *) (board.obj)
    6378,031   0,2     6378,031   0,2 33589092 MoveListGenerator::Next(void) (movelistgenerator.obj)
    6255,324   0,1     6255,324   0,1 21649383 Board::EvalMaterial(int) (board.obj)
    5267,619   0,1    15604,179   0,4   436050 MoveListGenerator::ComputeBlackKnightMoves(class Board *)
    5167,968   0,1    15984,870   0,4   970338 MoveListGenerator::ComputeBlackBishopMoves(class Board *,int)
    4747,808   0,1     4747,808   0,1 22468705 Board::~Board(void) (board.obj)
    4363,588   0,1   167127,750   4,0 10662437 CutoffHistory::SortMoveList(class MoveListGenerator *,int) (cutoffhistory.obj)
    4294,301   0,1     4294,301   0,1 24524291 Move::operator=(class Move const &) (movelistgenerator.obj)
    4207,342   0,1   162764,161   3,9 10662437 MoveListGenerator::Sort(void) (movelistgenerator.obj)
    3875,101   0,1    10595,647   0,3   316976 MoveListGenerator::ComputeWhiteKnightMoves(class Board *)
    3484,869   0,1    11232,858   0,3   637720 MoveListGenerator::ComputeWhiteBishopMoves(class Board *,int)
    3278,322   0,1    36286,634   0,9   729186 std::_Buffered_merge_sort(class std::reverse_iterator<class Move *,class
```

```
3014,502   0,1     7891,541   0,2   984663 MoveListGenerator::ComputeBlackRookMoves(class Board *,int)
2866,120   0,1     5864,081   0,1   730558 std::_Chunked_merge(class Move *,class Move *,class
2520,297   0,1     3542,164   0,1  5459781 std::copy(class std::reverse_iterator<class Move *,class Move,class Move
2271,786   0,1     6692,740   0,2   637171 MoveListGenerator::ComputeWhiteRookMoves(class Board *,int)
2242,235   0,1     2242,235   0,1 10662352 MoveListGenerator::ResetIterator(void) (movelistgenerator.obj)
2198,542   0,1    10572,962   0,3   421935 MoveListGenerator::ComputeBlackPawnMoves(class Board *)
1916,409   0,0  4159159,895  98,8        1 Game::RunGame(void) (game.obj)
1491,306   0,0    93569,910   2,2   819388 MoveListGenerator::ComputeLegalMoves(class Board *) (movelistgenerator.obj)
1384,970   0,0     8191,419   0,2   310093 MoveListGenerator::ComputeWhitePawnMoves(class Board *)
 856,485   0,0     2573,839   0,1   484825 MoveListGenerator::ComputeBlackKingMoves(class Board *) (
 851,825   0,0    16751,248   0,4  1460524 std::_Chunked_merge(class std::reverse_iterator<class Move *,class Move,class
 712,898   0,0     2485,645   0,1   318171 MoveListGenerator::ComputeWhiteKingMoves(class Board *)
  14,333   0,0  4154921,273  98,7       85 AISearchAgentNegaScout::UnrolledNegaScout(class Board *,int,class Move
   0,864   0,0  4156650,839  98,8       22 PlayerAI::GetMove(class Board *,class Move *) (playerai.obj)
   0,075   0,0  4154921,348  98,7       70 AISearchAgentNegaScout::NegaScoutDriver(class Board *,int,int,class Move *))
   0,004   0,0  4161046,424  98,9        1 _main (vschess.obj)
```

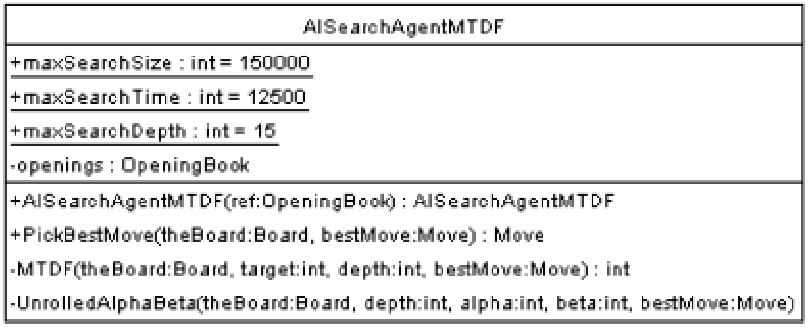
Am oben stehenden Profilerlauf (ein Profilerlauf zeigt die Zeit und Aufrufe an, die eine Funktion benötigt hat, alle Funktionen, die weniger als 0,1% der Rechenzeit benötigen sind nicht aufgelistet) kann man erkennen, dass der Computer über 90% der Zeit damit verbringt, die Ruhesuche zu bearbeiten. Dort müssten auch erste Optimierungsansätze stattfinden.
Durch genaueres studieren des Programmcodes mit Debugausgaben wird deutlich, dass die Zugsortierung in der Ruhesuche besser sein könnte. Ausserhalb der Ruhesuche wird zu ungefähr 95% der richtige Zug ausgewählt, der sofort einen Abbruch in der Suche bewirkt und somit Zeit spart. In der Ruhesuche ist der Erfolgsfaktor nur 70%, was deutlich in der Performance spürbar wird.
Zudem wird das *Transpositionsverzeichniss* noch nicht optimal genutzt. Es wird am Anfang jedes neuen Zuges der KI gelöscht (bis auf wenige Werte). Dies wäre eigentlich nicht nötig, doch waren die Regeln, nach denen das überschreiben alter Werte im *Transpositionsverzeichniss* fehlerhaft, so dass es jede Runde gelöscht wird. Dies wäre ein weiterer Ansatz zu optimieren.
Bisher schafft das Programm im Schnitt eine Suchtiefe von ca 6-7 Halbzügen in 10 Sekunden an einer 2 GHz CPU.
Mit oben genannten Erweiterungen sollten mindestens 7-8 Halbzüge möglich sein, was das Programm wesentlich stärker machen würde.

12.2 UML-Klassendiagramme

12.2.1 AISearchAgent

Die Factory AISearchAgent erstellt durch `MakeNewAgent()` einen KI-Spieler. Jeder einzelne Spieler wird einen unterschiedlichen Suchalgorithmus verwenden. `PickBestMove()` fordert die KI zum Ziehen auf.

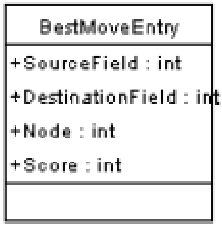
12.2.2 AISearchAgentMTDF



Der KI-Spieler AISearchAgentMTDF verwendet den MTD(f)-Suchalgorithmus (siehe „Aspirierte Suche und MTD(f)“). Die Suchtiefe kann über die `maxSearch` Variablen be-

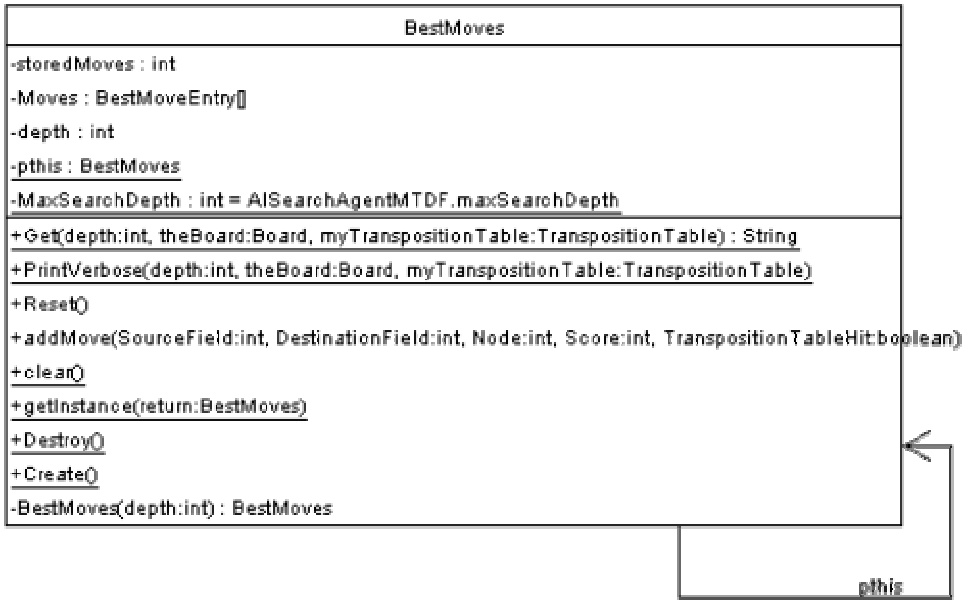
stimmt werden. Es wird sowohl die Eröffnungsdatenbank (siehe OpeningBook) als auch der Bewertungscache (siehe TranspositionTable) verwendet.

12.2.3 BestMoveEntry



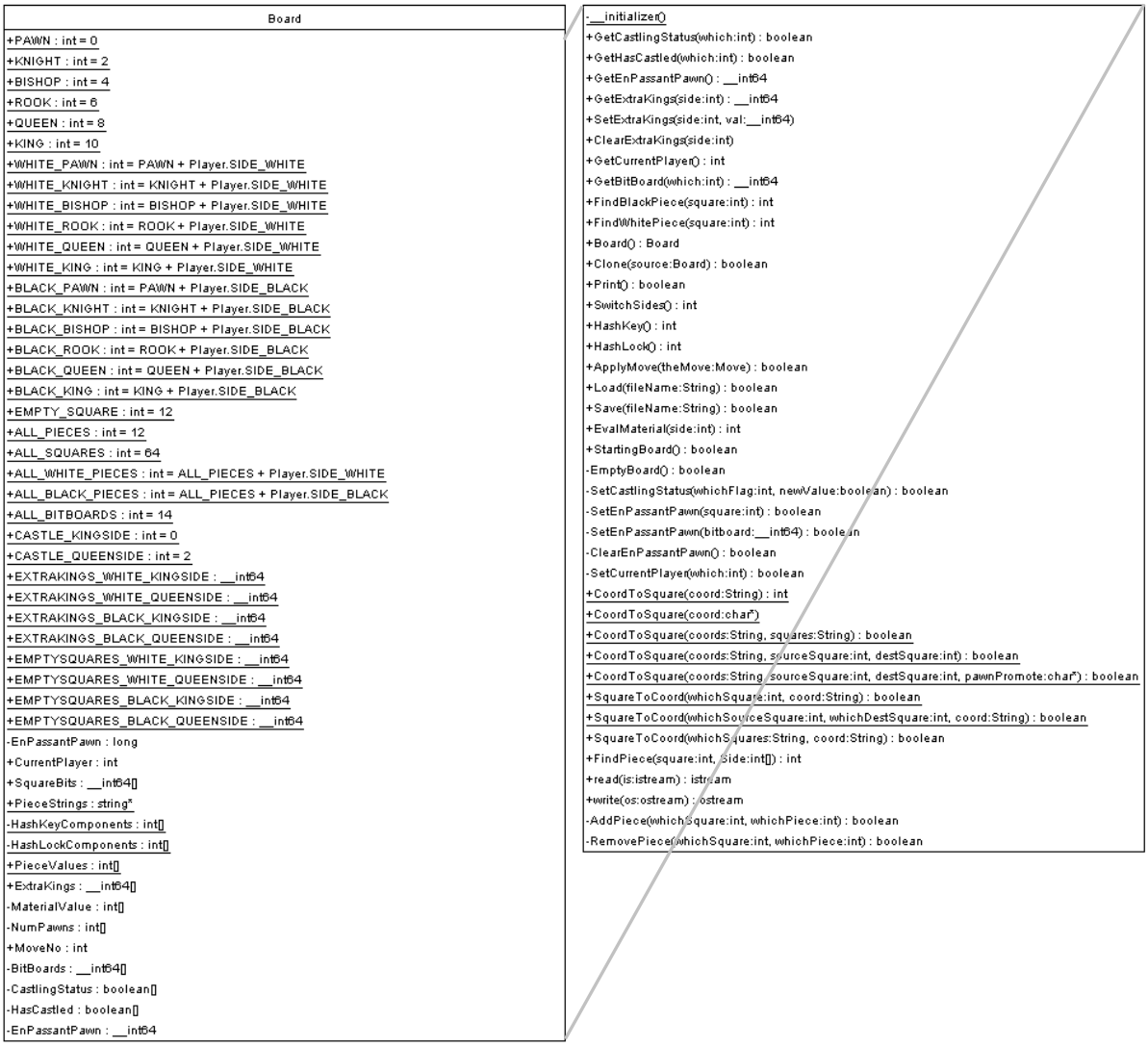
Speichert den Besten Zug des KI-Spielers für eine gegebene Suchtiefe. Somit kann verfolgt werden, was die KI denkt. Konkret heißt das, das der beste Zug für die jeweilige Seite gespeichert wird.

12.2.4 BestMoves



Die Klasse `BestMoves` dient zur Verfolgung der KI-Spieler. In ihr wird der beste Zug einer gegebenen Zugtiefe gespeichert und im DEBUG-Modus auch auf der Konsole ausgegeben. Weiterhin kann damit bei Verwendung von X/WinBoard ausgegeben werden, welche Züge der KI-Spieler momentan favorisiert.
Die besten Züge werden im Array `Moves` mit mehreren Instanzen der Klasse `BestMoveEntry` gespeichert.

12.2.5 Board

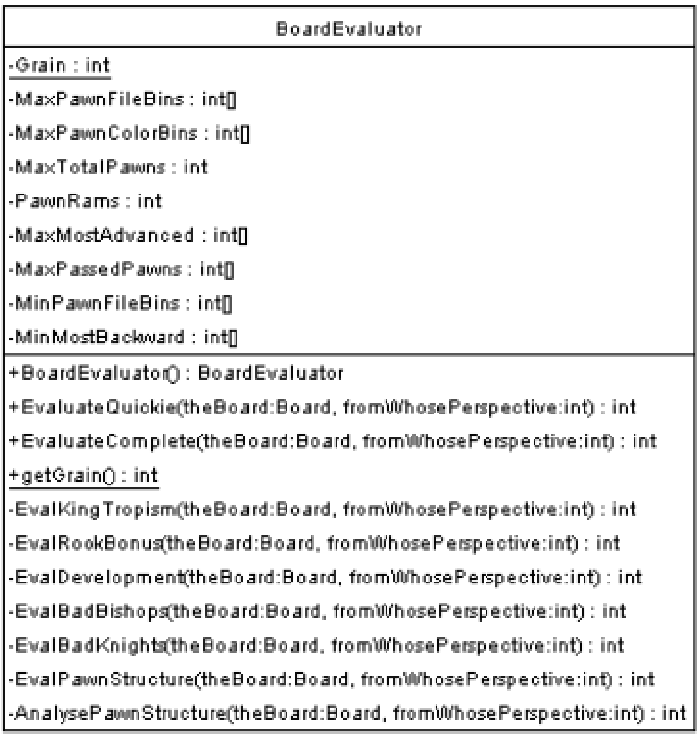


Die Boardklasse repräsentiert das Spielbrett. Jedes Spielfeld wird durch eine Zahl wir folgt repräsentiert:

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

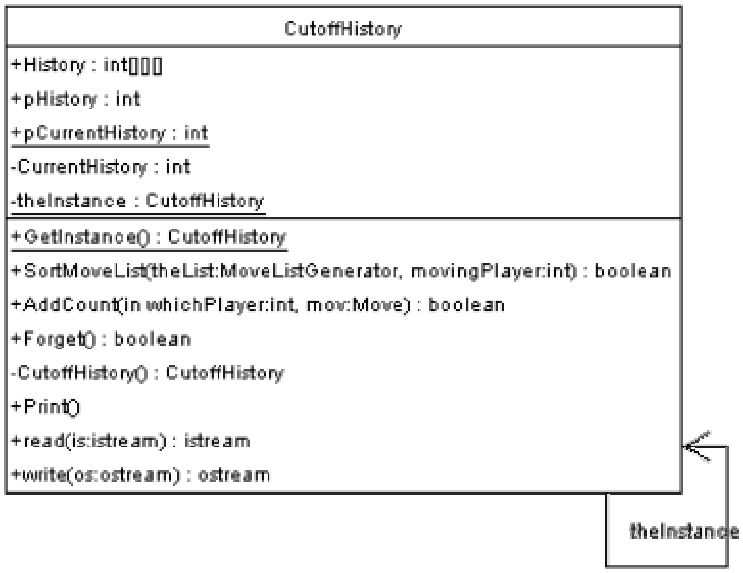
Die Information welche Figur wo steht, ist in `BitBoard[]` enthalten. Für jede Spielfigur ist dort eine 64-Bit Zahl abgelegt. Jedes Bit repräsentiert ein Spielfeld und wird auf 1 gesetzt, falls dort eine Spielfigur steht. Welches Bit welches Feld repräsentiert, kann in `Square-Bits[]` nachgeschaut werden.
Um aus der Feldnummer die (verständlichen) Koordinaten in Zeile und Spalte zu berechnen (oder umgekehrt) sollten die Funktionen `SquareToCoord()` und `CoordToSquare()` benutzt werden.

12.2.6 BoardEvaluator



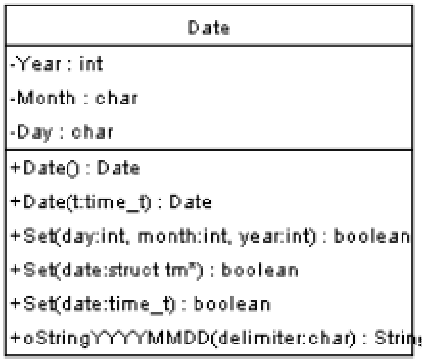
Implementiert alle die in Kapitel 8. Bewertungsfunktionen vorgestellten Bewertungen.

12.2.7 CutoffHistory



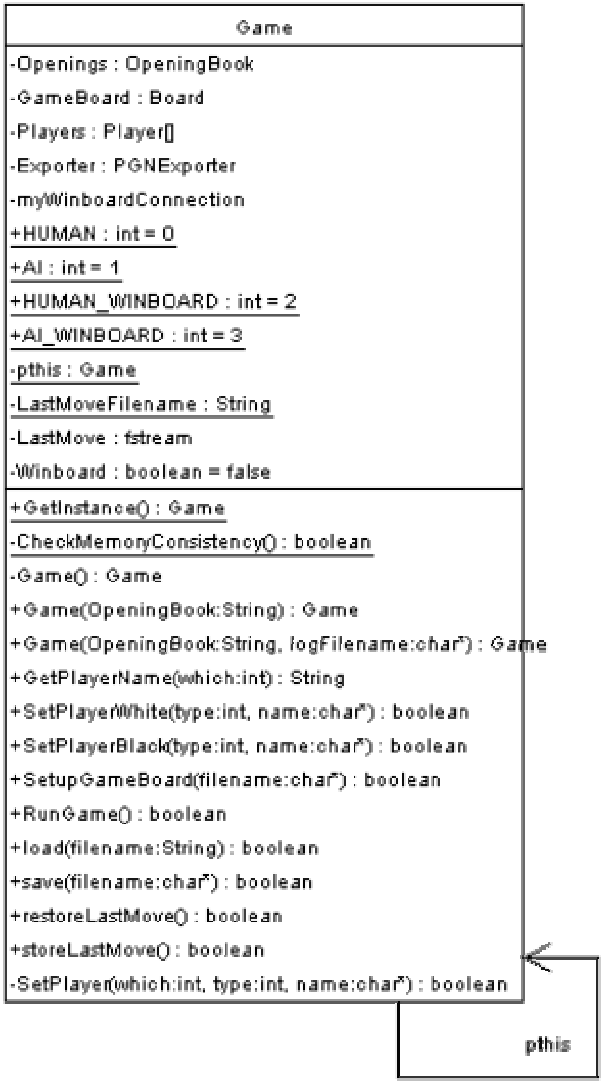
Diese Klasse speichert schon berechnete Züge, die einen Cutoff im Suchalgorithmus verursacht haben (vgl. Kapitel 6.4. Züge für den AlphaBeta vorsortieren). Dies soll helfen *Killer-Züge* zu identifizieren und anhand der Züge zu sortieren, bevor die aufwendige Bewertung gestartet wird.
Die mit einer bestimmten Wahrscheinlichkeit aufgerufene Methode `Forget()` sorgt dafür, dass nicht alte Züge einer alten Berechnung die aktuelle verfälschen.

12.2.8 Date



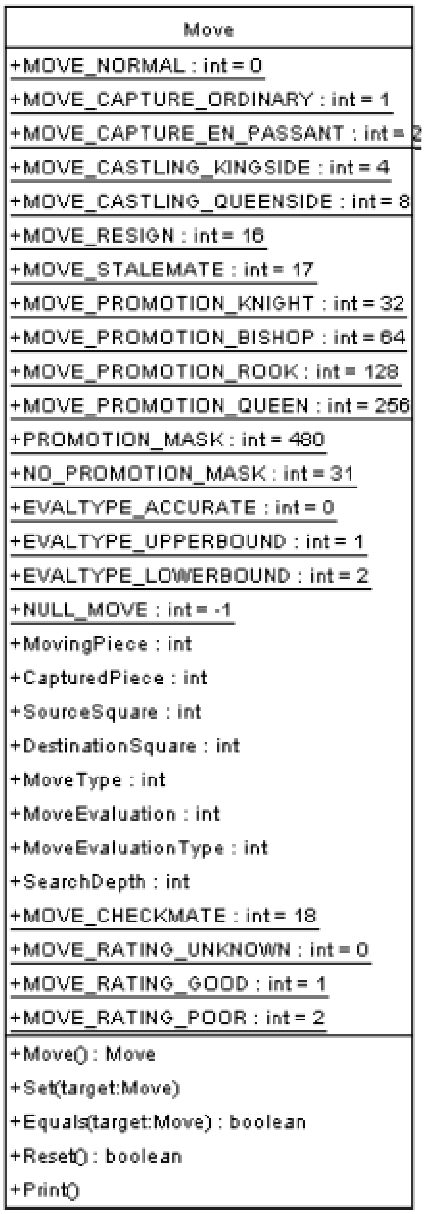
Date ist eine kleine Hilfsklasse, die die komfortable Verwaltung von Datumsangaben ermöglicht. Im Konstruktor kann die ANSI C Methode `time()` aufgerufen werden und schon ist das aktuelle Datum verfügbar.

12.2.9 Game



Die Klasse `Game` stellt das Applikationsframework für die Schachengine zur Verfügung. Hier werden die Spieler initialisiert, für jede Seite `PickBestMove()` aufgerufen und der gewählte Zug in die aktuelle PGN-Spieldatei kopiert.

12.2.10 Move

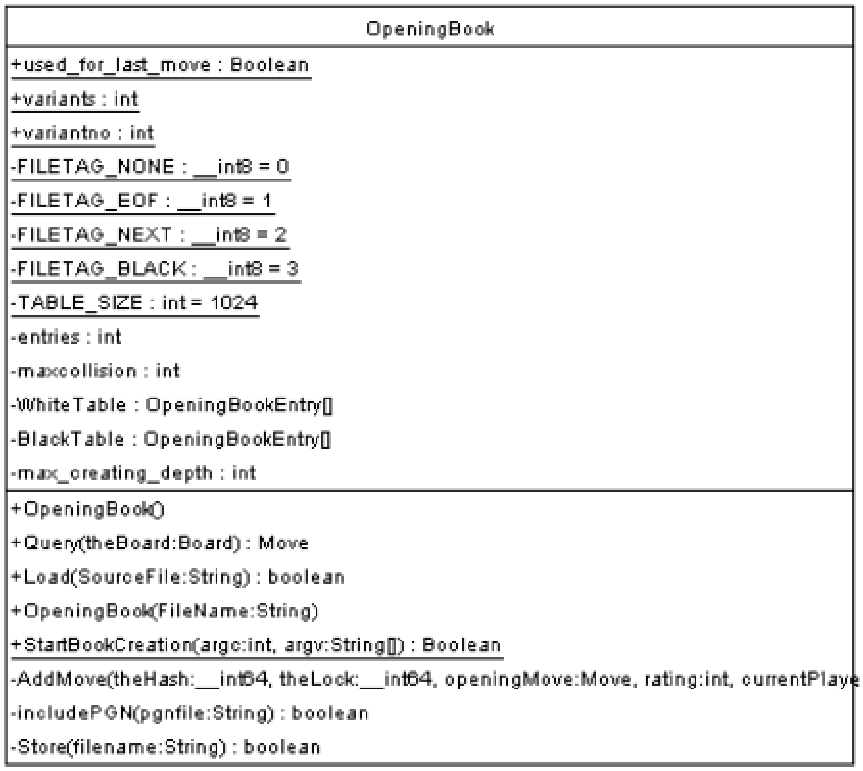


Diese Klasse repräsentiert einen Spielzug. Es wird das Start- und Zielfeld, die agierenden Figuren und die Zugbewertung gespeichert. Diese ergibt sich aus der Bewertung der entstehenden Brettsituation.
Gültige Spielzüge werden ausschließlich vom `MoveListGenerator` erzeugt.

12.2.11 MoveListGenerator

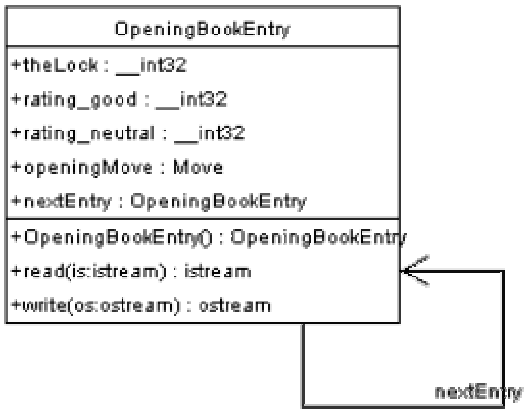
Die Klasse `MoveListGenerator` realisiert die Berechnung der Zugmöglichkeiten aller Figuren eines Spielers für eine gegebene Brettsituation. Alle der in Kapitel 5. Berechnung der Zugmöglichkeiten vorgestellten Berechnungen lassen sich hiermit verwirklichen.
Die Klasse besitzt eine riesige Look-up Tabelle, in der für jeden Figurtyp und seine Ausgangsposition seine möglichen Zielfelder gespeichert sind.
Der `MoveListGenerator` erzeugt ausschließlich gültige Züge. Falls eine eigene Figur den Weg blockiert, werden die dahintergehenden Felder übersprungen (z.B. Bauer steht vor Turm). Die erzeugten Züge sind vollständig und richtig (auch Rochaden und Umwandlungen werden berücksichtigt).

12.2.12 OpeningBook



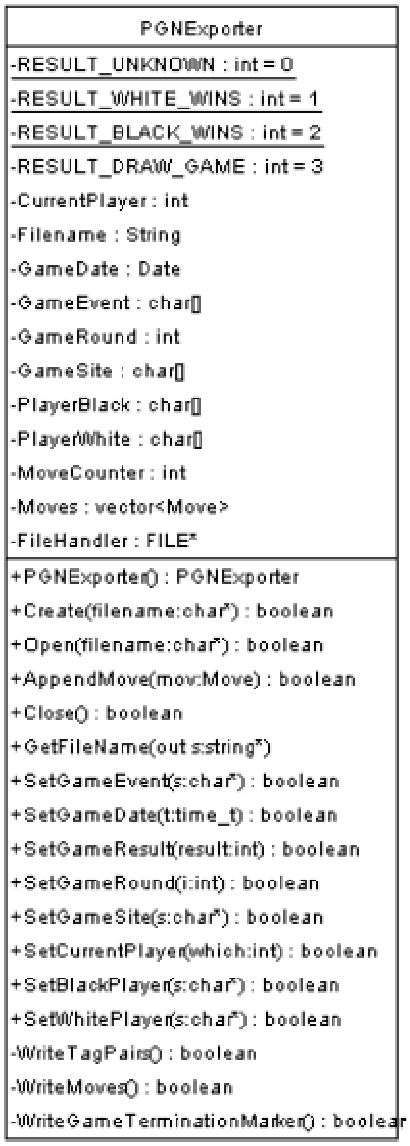
Das `OpeningBook` enthält eine Eröffnungsdatenbank, die der KI helfen soll, vor allem beim Beginn des Spiels (wo die Bewertungsfunktionen noch nicht so richtig greifen) sinnvolle Züge zu machen. Dafür wurden mit der Funktion `StartBookCreation()` 167.000 Turnierspiele geparst, die siegreichen und unentschiedenen Eröffnungen in die Datenbank aufgenommen und binär als Stream von `OpeningBookEntries` in der Datei `small-book.bin` gespeichert.

12.2.13 OpeningBookEntry



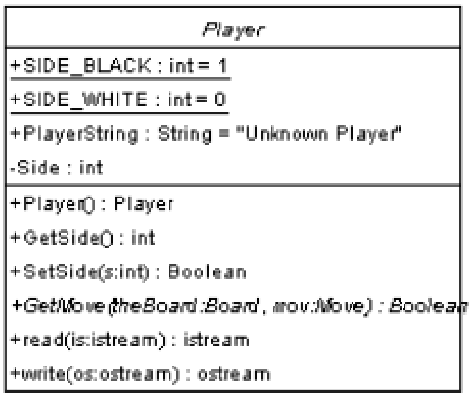
Enthält eine Zugvariante fürs `OpeningBook`. Die Brettsituation ist implizit durch die Reihenfolge im `OpeningBook` gegeben, aber sicherhaltshalber ist der Lock-Wert mitgespeichert. `NextEntry` is eine Zeiger zu unterschiedlichen Varianten einer Brettsituation.

12.2.14 PGNExporter



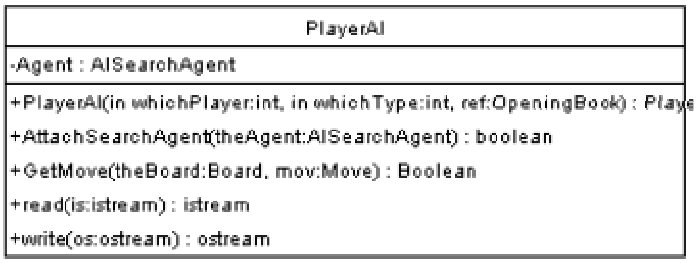
Diese Klasse dient dazu die aktuelle Partie in der *Portable Game Notation* (siehe Anhang Portable Game Notation) in eine Datei zu speichern. Der Standarddateiname ist `Games.pgn`. Das Spieldatum, die Spielernamen und das Ergebnis im *Seven Tag Roster* werden automatisch gesetzt. Mit *AppendMove* kann der gespielte Zug gespeichert werden.
Kommentare und das Spielen von Varianten werden nicht unterstützt.

12.2.15 Player



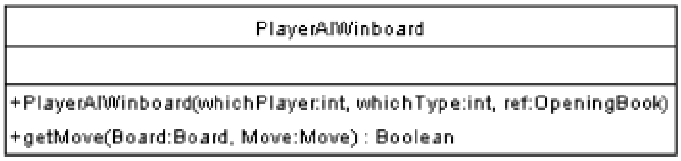
Die Klasse `Player` ist Superklasse sowohl für menschliche als auch KI-Spieler. Hier wird nur der Spielername gespeichert und festgelegt, dass die Funktion `GetMove()` implementiert werden muss. Ob dahinter sich nun konkret eine Tastatureingabe, eine Befehl vom X/WinBoard oder das Ergebnis einer MTD(f)-Suche verbirgt ist für diese Klasse (und für das Applikationsframework `Game`) uninteressant.

12.2.16 PlayerAI



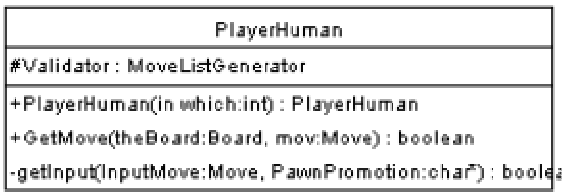
In `PlayerAI` stößt der Aufruf der Funktion `GetMove()` einen AISearchAgent an und startet den Suchalgorithmus.

12.2.17 PlayerAIWinboard



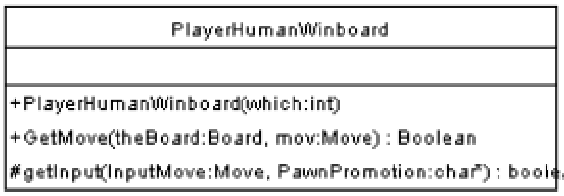
`PlayerAIWinBoard` ist eine Spezialisierung von `PlayerAI` und sorgt für die korrekte Kommunikation mit X/WinBoard über die Standard Ein- und Ausgabe.

12.2.18 PlayerHuman



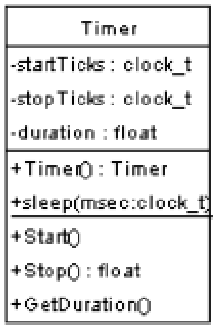
In `PlayerHuman` kann der Benutzer seinen Zug mit der Tastatur eingeben. Es wird überprüft, ob der Zug gültig ist und als Hilfestellung werden im Fehlerfall alle gültigen Züge ausgegeben, von denen sich der Spieler nur noch einen Zug aussuchen muss.

12.2.19 PlayerHumanWinboard



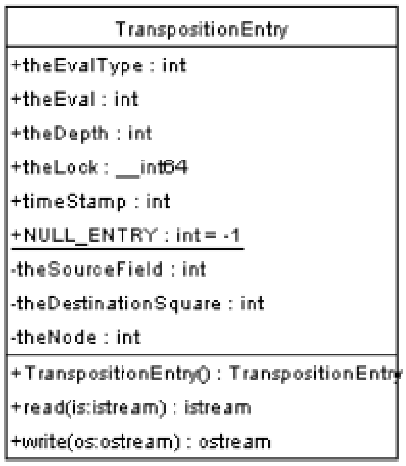
Diese Klasse ist eine Spezialisierung von `PlayerHuman` und sorgt für die korrekte Kommunikation mit X/Winboard über die Standard Ein- und Ausgabe.

12.2.20 Timer



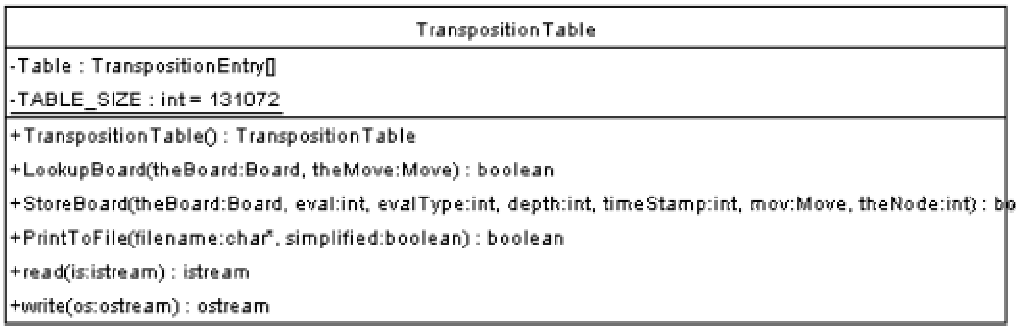
Diese kleine Hilfsklasse vereinfacht den Umgang mit Zeiten und stellt eine Stopuhr zur Verfügung. Mit Hilfe dieser Klasse wird die Performance der Chessengine gemessen.

12.2.21 TranspositionEntry



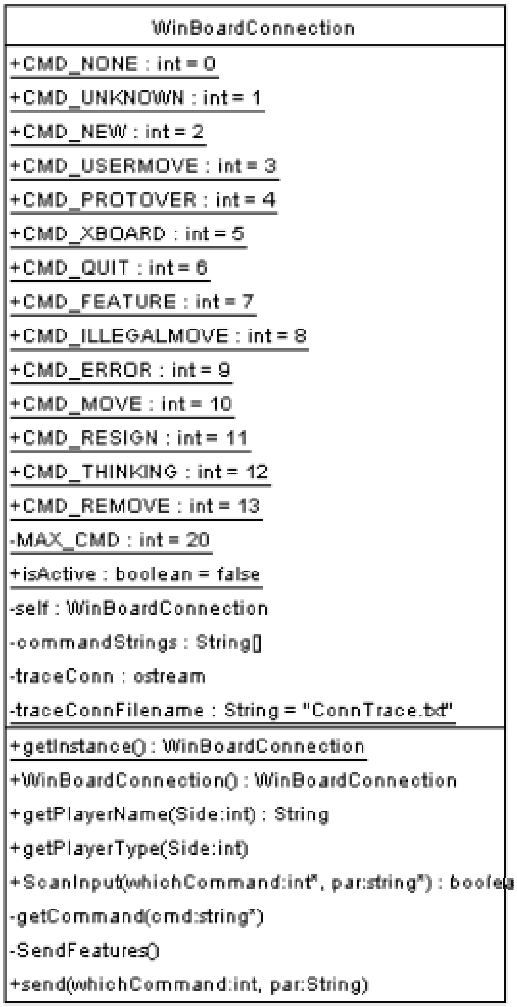
Die Klasse stellt einen Cacheeintrag in dem *Transpositionsverzeichnis* dar. Stellvertretend für die Brettsituation wird der *Hashwert* und der *HashLock* gespeichert und natürlich die zu cachende Bewertung.

12.2.22 TranspositionTable



Das Transpositionsverzeichnis cached Bewertungen für eine Brettsituation und erspart so eine aufwendige Neuberechnung. Sinnvoll ist dieser Cache deswegen, weil gleiche Brettsituation durch das Ziehen gleicher Figuren in einer unterschiedlichen Reihenfolge erzielt werden können. Der interessierte Leser kann die internen Statistiken in `AISearchAgent` (z.B. `numRegularTTHits`) auswerten.

12.2.23 WinboardConnection

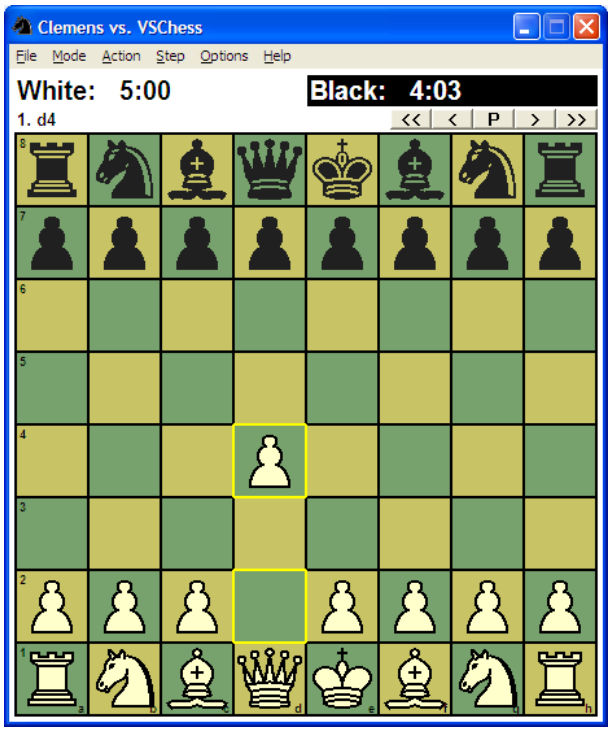


WinBoardConnection implementiert das Kommunikationsprotokoll von X/WinBoard. Der Nachrichtenaustausch erfolgt einfach über die Standardein- und ausgabe.
Mit `send()` kann eine Nachricht gesendet werden und einkommende Nachrichten werden gepuffert und können über `ScanInput()` gelesen werden.

12.3 X/WinBoard

Xboard und WinBoard sind graphische Benutzeroberflächen für Schach. Sie stellen das Schachbrett dar, akzeptieren Züge per Mauseingabe und laden und speichern im der *Portable Game Notation* (PGN). Die beiden Programme dienen als Front-ends für:

- Chess engines (GNU Chess, Crafty, BaChess und andere)
- Chess Server im Internet
- Schachspiele per E-Mail (Das Cmail Programm automatisiert das Parsen der E-Mail der Gegners, ziehen der Züge und versenden der E-Mail ihres gewählten Zuges)
- Ansehen gespeicherter Spiele (im PGN Format)



Weitere Informationen über den Autor und kompatible Chess-Server sowie Engines unter www.tim-mann.org. Die Software ist Freeware und mit Source-Code verfügbar.