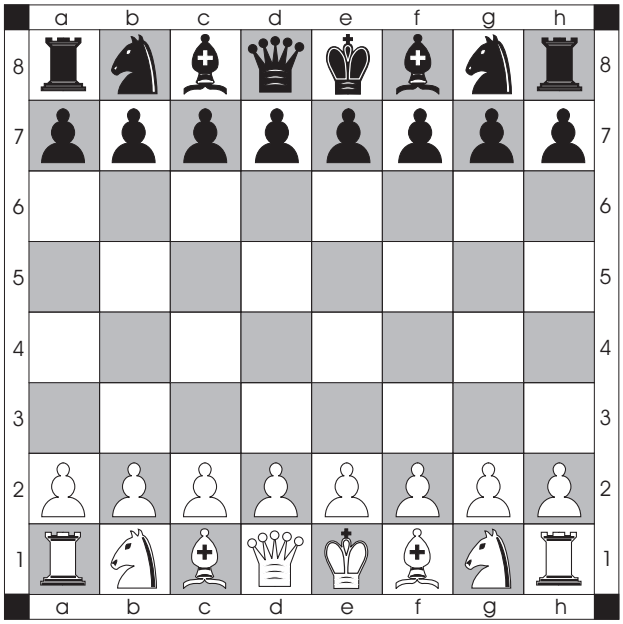
Betreuer: Prof. Dr. Schöning
Universität Ulm
Individualpraktikum
Sommersemester 07

Schachcomputer in C++



Michael Josef Staud
Josef-Strobel-Straße 15
88213 Ravensburg-Oberzell
staud@staudsoft.com
Informatik Diplom
8. Semster

Ravensburg, den 23. August 2007

Inhaltsverzeichnis

1 Einleitung

In diesem Praktikumsbericht wird die Entwicklung eines Schachcomputers beschrieben. Dabei stellt sich zuerst einmal die Frage, warum sollte man sich überhaupt mit Schach beschäftigen und dann einen Schachcomputer programmieren? Zuerst einmal ist Schach ein bekanntes Spiel. Fast jeder hat schon einmal Schach gespielt. Es verlangt mehr als jedes andere Spiel logischen Denken und Taktik. Somit wird auch der Verstand geschult. Manche sagen sogar, dass es dass interessanteste Brettspiel ist.

Deshalb gibt es auch sehr viele Schachcomputer. In der Tat war die Entwicklung von Schachcomputern eine der ersten Aufgaben der KI [1]. Leider ist aber das Angebot für PDAs im Moment schlecht. Ich habe keinen für mich zufriedenstellende Lösung gefunden. Deshalb entschied ich mich selber einen Schachcomputer zu entwickeln. Das wollte ich schon immer tun seit ich das erste mal mit einem Schachcomputer gespielt habe. Doch war ich bisher nicht in der Lage einen zu schreiben, da ich die erforderlichen Kenntnisse noch nicht erworben hatte. Erst durch das intensive beschäftigen mit Algorithmen und KI war mir klar, wie in diesem Fall vorzugehen ist. Dennoch war mein erster Versuch nicht funktionsfähig. In diesem Praktikumsbericht wird nun die zweite Version beschrieben. Ziel war es einen möglichst effizienten Schachcomputer mit wenig Codezeilen zu schreiben.

In Kapitel 2 dieses Berichts werden die für die Entwicklung eines Schachcomputers notwendigen theoretischen Grundlagen vermittelt. Es wird allgemein auf Schach eingegangen, dann wird die Suche nach einem möglichst optimalen Zug beschrieben. Dazu ist eine gute Bewertigungsfunktion unerlsslich. Sie wird in Kapitel 2.3 beschrieben. Um Sie möglichst gut zu optimieren, habe ich einen Ansatz des genetischen Programmierens gewählt.

In Kapitel 3 wird nun der tatsächliche Schachcomputer beschrieben. Zuerstmal wird der Aufbau der Objekte dargestellt, dann die Darstellung des Schachbretts. In darauffolgenden Kapitel wird die Zuggenerierung beschriebn, also wie aus den Regel von Schach alle möglichen Züge für eine bestimmte Position hergeleitet werden können. Ebenso wird erläutert wie auf Schach, Matt und Remis getestet wird. Danach wird die konkrete Implementierung des Suchalgorithmuses beschrieben und die Berechnung der Bewertungsfunktion mit der Herleitung. Als letztes werden Probleme bei der Portierung auf den PDA beleuchtet.

In Kapitel 4 werden die Leistungsfähigkeit und die Problem des Schachcomputers beschrieben. Ebenso werden Vergleichspartien mit anderen Schachcomputer aufgeführt. Kapitel 5 zeigt einen möglichen Ausblick.

2 Theorie

In diesem Kapitel geht es um die theoretischen Grundlagen, die für die Entwicklung eines Schachcomputers nötig sind. Zuerst wird Schach mit seinen Regeln beschrieben. Diese werden dann im praktischen Teil in den Quellcode eingebaut. Dann wird der Suchalgorithmus beschrieben, der entscheidet welcher Zug als nächstes zu wählen ist. Dabei werden auch neuere Optimierungstechniken beschrieben. Damit die Suche möglichst erfolgreich ist braucht man eine gute und schnelle Bewertungsfunktion. In Kapitel 2.3 wird beschrieben was bewertet wird und wie diese Bewertung realisiert wird. Und in Kapitel 2.4 wird ein Genetischer Algorithmus beschrieben.

2.1 Schach

Schach ist ein Spiel, in dem 2 Spieler gegeneinander spielen. Zu Beginn haben beide Spieler die gleiche Anzahl von Figuren. Ziel des Spiel ist es den gegnerischen König **Matt** zu setzen. Dass bedeutet, dass er bedroht wird durch eine andere Figur (er steht im **Schach**) und sich dieser Bedrohung nicht selber oder mit Hilfe anderer Figuren entziehen kann. Der *König* könnte also im nächsten Zug geschlagen werden. Dies wird aber nicht gemacht. Der *König* wird nie geschlagen. Stattdessen hat der Spieler verloren.

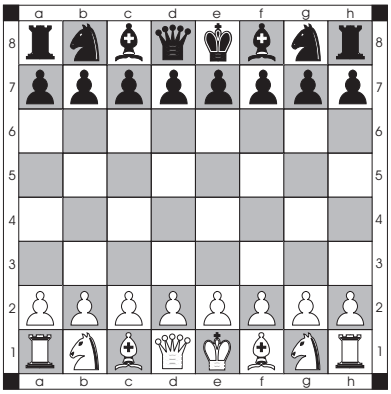


Abbildung 1: Schachbrett

Falls der *König* allerdings nicht im Schach steht und sich nicht mehr bewegen kann und der Spieler auch keine anderen Figuren hat endet die Partie unentschieden (**Remis**). Dies ist auch der Fall falls nicht mehr genügend Figuren vorhanden sind um die Gegenseite Matt zu setzen oder in 50 Züge kein Bauer gezogen wurde oder eine Figur geschlagen wurde.

Die Figuren im Schach werden nun verwendet um im Laufe einer Partie einen taktischen Vorteil zu erreichen und die Figuren des Gegenspielers zu reduzieren.

Im Schach gibt es 6 verschiedene Figuren: *König*, *Dame*, *Turm*, *Läufer*, *Springer*, *Bauer*. Jede Figur hat ihre eigene Art sich zu bewegen. Doch gibt es Gemeinsamkeiten. Allen Figuren außer der *Bauer* können genau die selben Felder betreten als die sie

schlagen können. Deren Bewegungsmuster kann man wieder weiterunterteilen in **Bewegungszüge** und **Springzüge**. **Bewegungszüge** sind durch eine Richtung und die maximale Anzahl der Schritte eindeutig bestimmt. Die Figur kann sich dabei soweit bewegen bis die maximale Anzahl der Schritte überschritten wird, eine gegnerische Figur geschlagen wird oder das weiterkommen durch eine eigene Figur behindert wird (alle Züge werden natürlich durch die Dimensionen des Schachbretts begrenzt). Somit kann man die Bewegungsart von *König*, *Dame*, *Läufer* und *Turm* ganz einfach durch ein Tupel aus Vektoren und der Maximalschrittanzahl bestimmen ($Vektor, Schrittanzahl$):

- *König*
  $((1,1,1),(1,-1,1),(-1,1,1),(-1,-1,1),$
  $(0,1,1),(0,-1,1),(-1,0,1),(-1,0,1))$
- *Dame*
  $((1,1,\infty),(1,-1,\infty),(-1,1,\infty),(-1,-1,\infty),$
  $(0,1,\infty),(0,-1,\infty),(-1,0,\infty),(-1,0,\infty))$
- *Turm*
  $(0,1,\infty),(0,-1,\infty),(-1,0,\infty),(-1,0,\infty))$
- *Läufer*
  $((1,1,\infty,(1,-1,\infty),(-1,1,\infty),(-1,-1,\infty))$

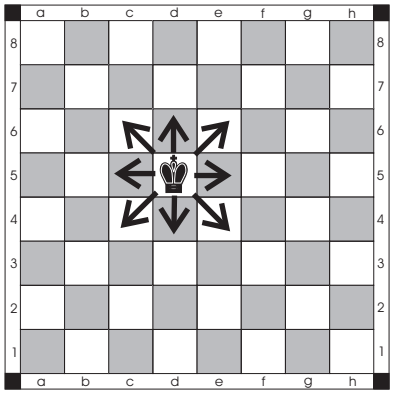


Abbildung 2: Bewegungsmöglichkeiten des Königs

Die Rösselsprünge des *Springers* gehören bei mir zur Kategorie der **Springzüge**. Diese werden durch einen Sprungvektor definiert. Der Zug kann vollzogen werden, falls sich auf dem Zielfeld keine eigene Figur befindet. Der *Springer* wird dabei nicht von anderen Figuren blockiert. Er überspringt sie einfach. Nun kann man auch das Bewegungsmuster des *Springers* durch ein Tupel von Vektoren beschreiben:
Springer:

$$((2,1),(1,2),(2,-1),(1,-2),(-2,1),(-1,2),(-2,-1),(-1,-2))$$

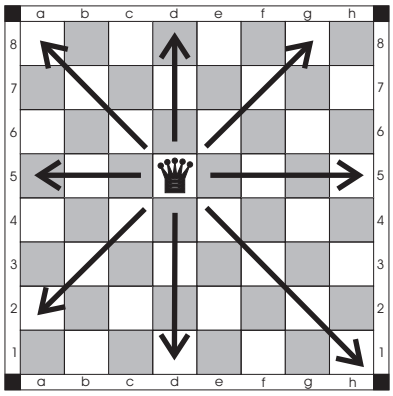


Abbildung 3: Bewegungsmöglichkeiten der Dame

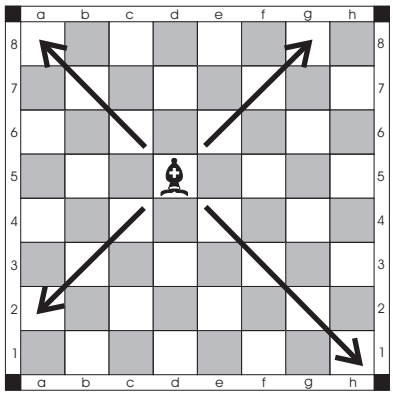


Abbildung 4: Bewegungsmöglichkeiten des Läufers

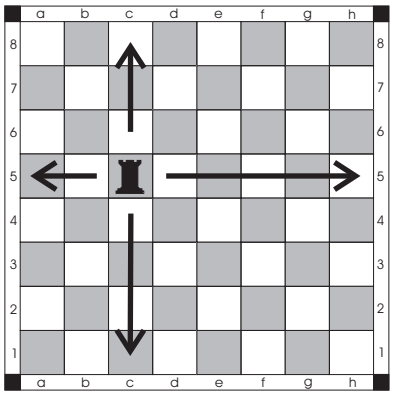


Abbildung 5: Bewegungsmöglichkeiten des Turms

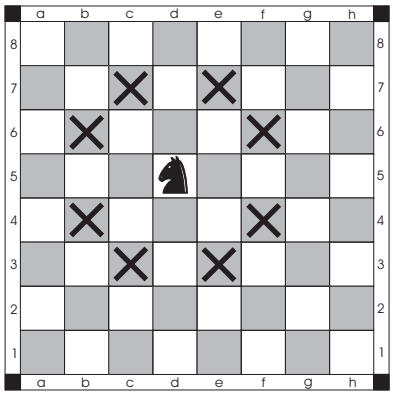


Abbildung 6: Bewegungsmöglichkeiten des Springers

Die letzte Figur die nun übrigbleibt, ist der *Bauer*. Ich habe ihre Bewegungsmuster den **Springzüge** zugeordnet. Der Bauer kann sich nur vorwärts in Richtung des Gegners bewegen und er kann nur seitlich nach vorne schlagen. Falls neben ihm ein Bauer steht kann er ihn auch **en passent** schlagen. Dies habe ich realisiert indem ich die **Springerzüge** um ein weiteres Feld erweitert habe. Diese zeigt an ob es ein Normaler (*Springer*), Move-Only, Move-Only-If-In-6-Row, Move-Only-If-In-2-Row, Beat-Only, Beat-Pawn-Only Zug ist. Mit dieser Erweiterung kann man nun den Bauer definieren:

- Weißer Bauer:
  $((0,-1,Move-Only),(0,-2,Move-Only-If-In-6-Row),(-1,-1,Beat-Only),(1,-1,Beat-Only),(-1,0,Beat-Pawn-Only),(1,0,Beat-Pawn-Only))$
- Schwarzer Bauer:
  $((0,1,Move-Only),(0,2,Move-Only-If-In-2-Row),(-1,1,Beat-Only),(1,1,Beat-Only),(-1,0,Beat-Pawn-Only),(1,0,Beat-Pawn-Only))$

Da der Bauer sich aus der Anfangsposition auch um 2 Felder bewegen kann wird bei dieser Konstellation noch zusätzlich getestet ob das übersprungene Feld frei ist.

! An den Vektoren für die *Bauern* kann man schon erkennen wie später intern das Schachbrett definiert ist. Die y-Achse geht von oben nach unten genau wie auf dem Computerbildschirm. Dies ist genau umgekehrt wie beim Schach.

2.2 Suche

Wie kann man nun einem Computer das Schachspielen beibringen? Das Programm muss sich bei jedem Zug überlegen, welcher der Beste ist und ihn ausführen. Doch wie kann man feststellen welcher Zug gut ist? Dies geschieht, indem man alle möglichen Züge generiert. Dann erzeugt man alle möglichen Züge die der Gegner machen kann usw. Es entsteht dabei ein endlicher Suchbaum. Könnte man alle Züge bis zum Ende des

Spiels berechnen, so könnte man einen Schachcomputer konstruieren der optimal spielt. Man würde alle Endknoten (die Matt oder Remis sind) in denen der Spieler gewonnen hat mit 1 markieren, alle Verlorenen mit -1 und alle Unentschieden mit 0. Dann würde man den Baum von unten nach oben mit dem MinMax Prinzip ([1]) abarbeiten. An jedem Knoten, an dem der Gegenspieler am Zug ist, übernimmt man den Wert des minimalen Knotens, weil man davon ausgeht, das der Gegenspieler optimal spielt und deshalb immer den besten Zug nimmt. Ist das nicht der Fall, kann er nur schlechter spielen was die Gewinnchancen des Schachcomputers erhöht. An den Knoten an dem der Schachcomputer am Zug ist maximiert man den Wert. So könnte der Schachcomputer optimal spielen. Er würde niemals verlieren, wenn das Spiel fair ist (man könnte durch so eine Analyse auch feststellen in wie weit Schwarz einen Nachteil hat, weil es erst als Zweiter ziehen darf).

Nun ist es aber mit heutigen Computer und wahrscheinlich auch mit Zukünftigen nicht möglich den Suchbaum bis zu den Blättern berechnen. Deshalb muss man die Suche irgendwann abbrechen, wenn gewisse Kriterien erfüllt sind. Dies kann z.B. der Fall sein, falls die maximale Tiefe überschritten wird oder man sieht, dass der Zug sehr schlecht war. Diese Knoten nennt man Horizontknoten. Man verwendet dann eine Bewertungsfunktion um den Rest der Suche abzuschätzen. Da man nicht genau sagen kann, ob ein Zug zum Sieg führt, gibt man jedem Spieler je nach Figurenanzahl und Stellungsvorteil Punkte. Die zieht man dann voneinander ab. Deshalb hat man dann auch andere Werte als -1,0, 1. Wie diese Funktion berechnet wird, wird in Kapitel 2.3 gezeigt. Dann führt man den normalen MinMax Algorithmus aus. Um nicht alle Knoten speichern zu müssen führt man eine Tiefensuche durch.

2.2.1 Alpha Beta

Damit der Schachcomputer so gut wie möglich wird, will man eine möglichst große Suchtiefe erreichen. Deshalb ist es sinnvoll den Schachcomputer noch etwas zu verbessern. Dies ist mit dem Alpha-Beta Algorithmus möglich ([1], [7]). Es basiert auf der Beobachtung, dass man z.T. die Berechnung von Knoten abbrechen kann, wenn sie in einem Elternknoten sowieso verworfen werden (siehe Abbildung 7). Das ist der Fall wenn, z.B. der aktuelle Knoten ein Max Knoten und der Elternknoten ein Min Knoten ist. Dann wurden im Elternknoten schon andere Knoten ausgewertet, sodass dort ein vorläufiges Minimum existiert. Wird nun der Wert des Max Knoten größer als das vorläufige Minimum, so ist klar, dass man ihn im Elternknoten verwerfen würde. Er braucht deshalb nicht zu Ende berechnet werden. Dies nennt man Cut-Off. Natürlich ist das ganze auch bei einem Min Knoten mit einem Max Elternknoten möglich.

Deshalb übergibt man dem Algorithmus das vorläufige Suchfenster $[\alpha, \beta]$. Dieser bricht die Berechnung sofort ab, wenn das Suchfenster verlassen wird. Durch diese Effekt reduziert sich die Anzahl der Äste, die man an einem Knoten untersuchen muss und man kann deshalb tiefer suchen.

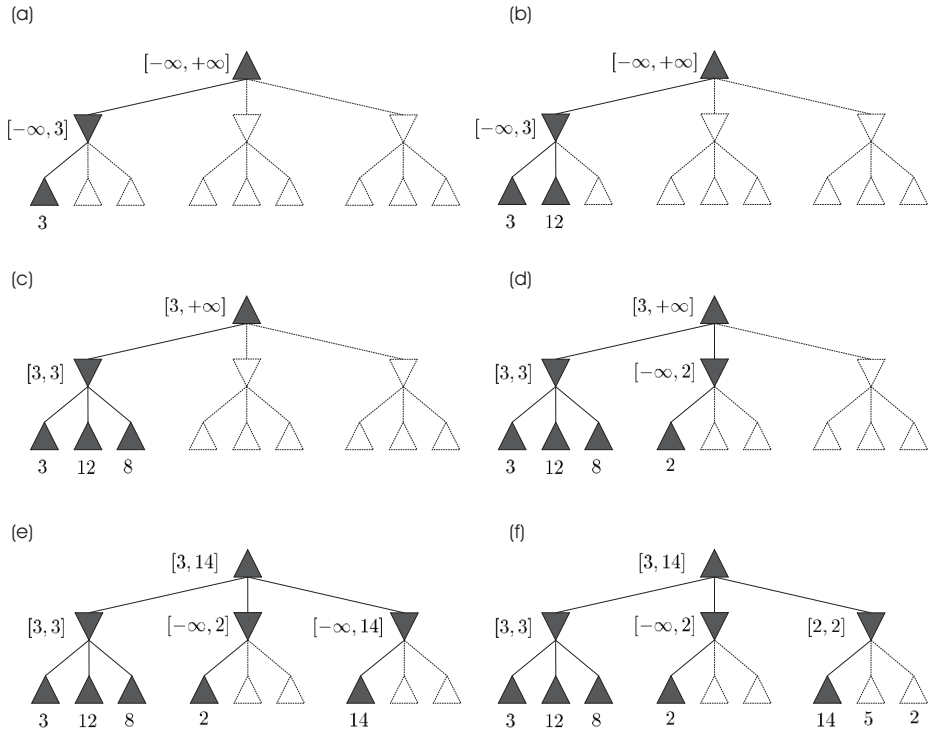


Abbildung 7: Alpha Beta
Dieses Beispiel des Alpha Beta Algorithmus wurde aus [1] übernommen. Man kann Schritt für Schritt sehen, wie die oberen und unteren Grenzen angepasst werden und deshalb Knoten ausgelassen werden.

2.2.2 Minimal Window Search

Eine weitere Möglichkeit ist es, nachdem man ein vorläufigen Minimum oder Maximum hat, nur noch ein minimales Suchfenster zu verwenden. Falls der Knoten ein Max Knoten ist, dann verwendet man ein Suchfenster $[\alpha, \alpha + 1]$. Die Idee dabei ist nur zu testen, ob man mit diesem Zug das Maximum noch verbessern kann. Dies geht durch die frühen Cut-Offs, die in einem so kleinen Suchfenster stattfinden relativ schnell. Falls das Ergebnis der Suche gleich α ist, braucht man keine genauere Suche mehr vornehmen, weil der Wert des Maximums nicht vergrößert wird. Falls das Ergebnis gleich $\alpha + 1$ ist, so hat sich der Wert vergrößert. Man muss deshalb noch eine normale Suche starten um den korrekten Wert zu berechnen. Dies wird auch in [3] beschrieben.

2.2.3 Quiescence Suche

Wie oben erwähnt muss man die Suche irgendwann abbrechen. Falls dies der Fall ist kann es zum Horizon Effekt kommen. Dieser besagt, dass man nichts über die weiteren Auswirkungen (in größeren Tiefen als die Suchtiefe) eines Zuges aussagen kann. Es kann z.B. sein, dass der Zug der bisher gut aussieht später zu einem Disaster wird, indem Sinne, dass der Spieler wichtige Figuren verliert oder sogar matt geht. Um das zu vermeiden verwendet man die sogenannte Quiescence Suche. Sie durchsucht nicht mehr die volle Bandbreite aller möglichen Züge sondern nur noch wenige Entscheidende. Es werden dann so viele neue Knoten durch diese Suche erzeugt bis man auf Position trifft, die gut durch die Bewertungsfunktion verwertbar ost. Durch diesen Trick kann man den Horizon Effekt abschwächen.

Doch welche Züge sollte man wählen? Man muss dabei einen Ausgleich zwischen Suchzeit und Qualität der Auswertung finden. Normalerweise führt man nur noch die Züge aus, in denen eine Figure geschlagen wird. Um die Geschwindigkeit weiter zu erhöhen sortiert man die Züge so, dass zuerst die untersucht werden, in denen eine möglichst wertvolle Figur der Gegenseite von einer möglichst wertlosen des Spielers geschlagen wird. Falls man dabei feststellt, dass man einen zu großen Materialverlust hat, kann man die Suche auch abbrechen (siehe [1]).

Durch diese Methoden kann man nun die Qualität der Bewertung der Horizontknoten verbessern.

2.2.4 Transposition Table

Im Schach kann man eine bestimmte Schachposition mit verschieden Züge erreichen. Deshalb findet die Suche nach einem Zug in einem azyklichen gerichteten Graphen statt. Es ist also sehr wahrscheinlich, dass der Alpha-Beta eine bestimmte Schachposition mehrmals auswertet, was sehr viel Zeit kostet. Deshalb verwendet man eine sogenannte Transposition Table. Sie ist eine Hashmap in der sich Schachpositionen befinden. In ihr wird ein Teil der berechneten Züge gespeichert. Und zwar speichert man die aktuelle Position, die Suchtiefe, den Wert und ob der Wert exakt ist oder eine untere bzw. obere Schranke ist.

Bevor man nun einen Zug auswertet schaut man in der Transposition Table nach. Ist der Zug vorhanden, testet man, ob die Suchtiefe des gefunden größer ist als die Aktuelle. Wenn ja, kann man die Daten verwenden um entweder direkt ein Ergebnis zu liefern oder um α und β weiter einzuschränken um somit Cut-Offs zu erzielen (siehe [1]).

2.2.5 Move Ordering

Um nun einen schnellen Cut-Off zu erreichen empfiehlt es sich, die Züge zuerst zu untersuchen, die am erfolgversprechenden sind. Dazu muss man sie nach bestimmten Kriterien sortieren. Dies kann im einfachsten Fall mit Hilfe der Bewertungsfunktion für die Horizontknoten gemacht werden. Dabei untersucht man zuerst die am besten bewerteten Knoten. Je nach dem, ob die Bewertungsfunktion dafür gut "geeignet" ist, erhält man bessere oder schlechtere Resultate. In meinem Schachcomputer befindet sich deshalb meistens der Beste Zug unter den ersten drei.

2.2.6 Iterative Deepening

Da man für jeden Zug nur eine begrenzte Zeit zur Verfügung hat, bietet es sich an, zuerst einmal mit einer kleineren Tiefe einen Zug zu berechnen und dann, falls man noch Zeit hat, den Suchalgorithmus nochmal mit einer größeren Tiefe auszurufen. Dies hat mehrere Vorteile. Erstmal werden die Informationen, die in der Transposition Table gespeichert wurden, in einer Suche mit einer größeren Tiefe nochmal verwendet. Es ist sogar so, dass es schneller ist, stückweise die Tiefe zu erhöhen, als einmal sehr tief zu suchen. Dann kann man aus der Zeit, die der Algorithmus gebraucht hat, um den Zug bis zur Tiefe t zu berechnen abschätzen, wieviel Zeit er für die Tiefe t+1 braucht. So kann man entscheiden, ob es sich lohnt noch tiefer zu suchen. Dies könnte z.B. der Fall sein, wenn die eigene Position sehr schlecht ist, also man am verlieren ist. Ebenso kann man auch eine Suche abbrechen, falls die Zeit ausgeht. Dann nimmt man einfach das Ergebnis einer vorherigen Suche.

2.2.7 Aspiration Search

Normalerweise beginnt man bei der Alpha-Beta mit einem unendlichen großen Suchfenster. Falls es gelingen würde, dieses Suchfenster einzugrenzen, könnte man die Geschwindigkeit erhöhen. Dazu muss man aber das Ergebnis der Suche abschätzen können und dann in einem Intervall um diese Schätzung suchen. Den geschätzten Wert bekommt man entweder aus der Suche vom vorherigen Zug oder, falls man Iterative Deepening verwendet, aus einem Zug mit einer geringeren Tiefe.

Falls man jedoch das Suchfenster zu klein wählt hat, muss man noch einmal mit einem unendlich großen Suchfenster suchen. Deshalb gilt es hier ein gutes Intervall zu nehmen. Üblicherweise nimmt man dabei den Wert eines halben Bauern.

2.3 Bewertungsfunktion

Wie schon oben bei der Suche erwähnt, ist es notwendig, die sogenannten Horizontknoten zu bewerten. Dabei bestimmt die Qualität der Bewertungsfunktion wie gut der ausgewählte Zug ist. Deshalb braucht man eine möglichst gute Bewertungsfunktion. Braucht sie allerdings zu lange mit der Bewertung, so reduziert sich wiederum die möglich Suchtiefe, was wiederum die Qualität beeinträchtigt. Man muss deshalb einen guten Mittelwert finden.

Wie bewertet man aber nun ein Schachfeld? Glücklichweise haben Schachspieler schon immer versucht ein Schachbrett zu bewerten. Es gibt deshalb in der normalen Schachliteratur viele Information darüber.

Aus [5] stammen die folgenden Bewertungen für die einzelnen Schachfiguren:

Figure	Wert
Bauer	100
Läufer	300
Springer	300
Turm	450
Dame	900
König	100000

Dem König habe ich selbst einen so großen Wert zugewiesen, damit eine Schachposition, die Matt ist, eine sehr gute Bewertung beim nächsten Zug bekommt. Bei der Suche wird also der König tatsächlich geschlagen. Da aber nach jedem Zug auf Matt getestet wird, tritt dieser Fall auf dem sichtbaren Spielfeld nicht auf.

Im Schachcomputer sind nun eine Reihe von Funktionen definiert, die bestimmte Konstellationen auf dem Schachfeld bewerten.

- Entfernung zum König: Ein Schachposition wird positiv gewichtet, falls die Dame und die Turm nahe am gegnerischen König sind.
- Mobilität: Für jede Seite wird abgezählt wieviele Felder durch die eigenen Figuren erreicht wird. Dies ist die sogenannte Mobilität. Ebenso wird es positiv bewertet, wenn gegnerische Figuren angegriffen werden.
- Bauernposition: Jeder Bauer bekommt Extrapunkte je näher er an die Gegenseite heranrückt. Freistehende Bauern bekommen mehr Punkte und Doppelbauern werden negativ bewertet.

Diese Funktionen werden zu einem einzigen Wert verrechnet.

2.4 Genetischer Algorithmus

Im letzten Kapitel wurde die Funktionen betrachtet, die bestimmte Aspekte auf dem Schachfeld bewerten. Diese Funktionen werden gewichtet und zu einem Wert zusammengefasst. Um eine möglichst optimale Gewichtung zu erreichen wird ein genetischer Algorithmus verwendet um diese zu bestimmen.

2.4.1 Algorithmus

Der allgemeine Genetische Algorithmus erzeugt am Anfang eine Population von m Individuen mit zufälligem Genetischem Code. Dann werden die Generationsschritte simuliert. Dabei wird zuerst eine Menge von zufälligen Kreuzungen erzeugt. Diese werden dann mutiert. Dann findet die Bewertung der Fitness statt. In Fall eines Schachcomputers spielen zwei verschiedene Gewichtungsfunktion gegeneinander. Die Gewinnende bekommte einen Punkt. Die Fittesten haben dann die meisten Punkte. Dann wählte man die m Fittesten aus. Diese bilden dann die nächste Generation. Das ganze wiederholt man solange bis man keine nennenswerten Verbesserungen der Fitness mehr erreicht. Im Fall eines Schachcomputers ist dies aber nicht möglich, da die Fitness relativ zu den anderen Individuen berechnet wird. Deshalb hört der Algorithmus entweder nach n Schritten auf, oder es wird beobachtet, ob die Anzahl der Remis steigt und die Fitness zwischen den Individuen immer mehr zur einer Gleichverteilung wird.

2.4.2 Genetischer Code

Der genetische Code besteht im Fall eines Schachcomputers aus einer Menge von ganzen Zahlen. Jede Zahl repräsentiert die Gewichtung einer Funktion:

- *int attackertable[6][6]*: Tabelle, die Punkte vergigt, wenn eine Figur mit einem bestimmten Typ eine andere mit einem bestimmten Typ angreift.
- *int dame_value*: Punkte, die die Dame für die Nähe zum gegnerische König bekommt.
- *int turm1_value*, *int turm2_value*: Punkte, die die Türme für die Nähe zum gegnerische König bekommen.
- *int springer1_value*, *int springer2_value*:Punkte, die die Springer für die Nähe zum gegnerischen König bekommen.
- *int laufer1_value*, *int laufer2_value*:Punkte, die die Läufer für die Nähe zum gegnerischen König bekommen.
- *int figure_value*: Wie stark das Materialgewicht beachtet wird.
- *int pawn_value*: Bewertung der Position der Bauern zur feindlichen Linie.
- *int chess_value*: Punkte, die man bekommt, wenn man den gegnerischen König in das Schach setzt.
- *int two_pawns*: Punkte, die man für Doppelbauern abgezogen bekommt
- *int free_pawn*: Punkte, die man für freie Bauern bekommt
- *int backward_pawn*: Punkte, die man für rückständige Bauern abgezogen bekommt

- *int mobility_value[6]*: Wieviel Punkte es bringt, wenn sich ein bestimmter Figurentyp auf dem Schachfeld bewegen kann
- *int notattacked_value[6]*: Zusätzliche Punkte für ein Feld, auf der sich die Figur bewegen kann ohne angegriffen zu werden.

Die Kreuzung von 2 Individuen wird mit uniform-crossingover ([3]) realisiert. Dabei operiert der Algorithmus allerdings nicht auf Bitebene, sondern auf sich jeweils entsprechenden Zahlenpaaren von 2 Individuen.

3 Schachcomputer

In diesem Kapitel wird nun beschrieben wie der Schachcomputer praktisch aufgebaut wird. Zuerst wird der Aufbau der Klassen beschrieben. Dann wird die interne Darstellung des Schachbretts betrachtet. Im nächsten Kapitel wird erläutert wie mit Hilfe von C++ Templateprogrammierung die Spielregeln effizient implementiert werden. Im Anschlußdaran wird der Suchalgorithmus beschrieben und danach die Bewertungsfunktion.

3.1 Aufbau

Bei der Entwicklung des Schachcomputers gab es für mich mehrere Ziele. Und zwar sollte es möglich sein, dass sowohl 2 Schachcomputer als auch 2 Menschen oder ein Mensch gegen einen Schachcomputer spielen kann. Dann wollte ich es ermöglichen, dass der Schachcomputer nachdenken kann und gleichzeitig noch Benutzereingaben verarbeiten kann. Deshalb wurde die Möglichkeit angelegt das Denken der KI in einen Thread auszulagern. Daraus ergibt sich dann die Architektur aus Abbildung 8.

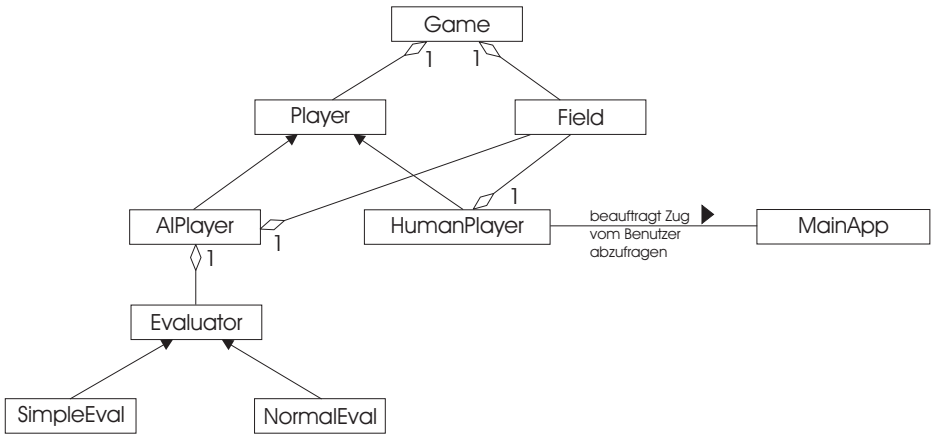


Abbildung 8: Klassenhierarchie

Zuerst einmal zum Spieler. Wie man sieht wird in den relevanten Stellen in der Klasse Game die allgemeine Spielerklasse verwendet. Dies ermöglicht es vollkommen frei festzulegen, ob ein Spieler ein Mensch oder ein Computer ist. Was aus diesem Diagramm nicht ersichtlich ist ist, dass die Oberfläche mit dem HumanPlayer zusammenarbeitet. Wird dieser aufgefordert einen Zug zu berechnen, informiert er die Oberfläche darüber, dass nun Benutzereingaben für die entsprechende Seite erlaubt sind. Macht der Spieler einen Zug testet die Oberfläche, ob dieser erlaubt ist und teilt ihn dann dem HumanPlayer mit. Dieser führt dann den Zug aus. Der AIPlayer arbeitet für sich allein.

Den Abblauf eines Zuges kann man in dem Interaktionsdiagramm in Abbildung 9 sehen.

Zuerst wird entweder vom Anwendungsprogramm oder von Update die Methode *Game::MakeDraw()* aufgerufen. Dies ist die Aufforderung einen neuen Zug zu berechnen.

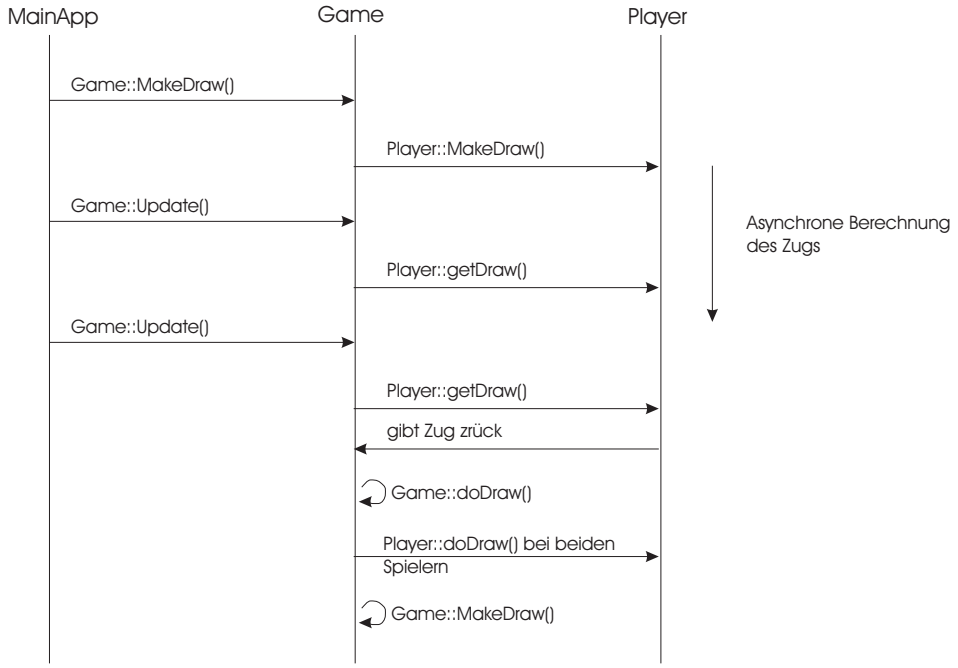


Abbildung 9: Interaktionsdiagramm bei der Berechnung eines Zuges

Game::MakeDraw() ruft nun *Player::MakeDraw()* vom dem Spieler auf, der aktuell am Zug ist. Dieser fängt nun an asynchron einen Zug zu berechnen bzw. bei *HumanPlayer* die Benutzereingabe zu ermöglichen. Das bedeutet *Player::MakeDraw()* kehrt sofort zurück und *Game::MakeDraw()* ebenso. Von außen (konkret aus der Klasse *MainApp* mit einem Timer) wird nun in regelmäßigen Abständen *Game::Update()* aufgerufen. Diese Funktion ruft *Player::getDraw()* auf. Diese liefert *NULL* zurück, falls der Zug noch nicht fertig berechnet wurde. Dann kehrt *Game::Update()* mit false zurück. Falls hingegen ein Zug zurückgeliefert wird ist der Player mit der Zugberechnung fertig. Dieser Zug wird nun mit *Game::doDraw* auf den internen Schachfeldern ausgeführt (in Game und den Playern). Falls es nicht zu einem Matt oder Remis gekommen ist, wird mit *Game::MakeDraw()* die Berechnung des nächsten Zuges angefordert. Es wird true zurückgegeben.

Wie oben erwähnt führt *Game::doDraw* nun den Zug aus. Danach wird auf Matt und Patt getestet. *Game::gamestate* enthält den momentanen Zustand des Spiels: *gsPlaying, gsBlackWon, gsWhiteWon, gsRemis*. Dieser wird dann in *Game::doDraw* angepasst. Ebenso wird die Berechnung eines neuen Zuges nicht mehr in *Game::Update()* angefordert, falls *Game::gamestate* nicht *gsPlaying* ist.

Für Debuggingzwecke (und ebenso für den Genetischen Algorithmus) kann man die asynchrone Berechnung des Zuges im *AIPlayer* ausgeschaltet. So kehrt *Player::MakeDraw()* wirklich erst nach der Berechnung des Zuges zurück. Dies beeinflusst allerdings den obigen Ablauf nicht negativ.

3.2 Schachbrett

Das Schachbrett wird in Schachcomputern normalerweise durch ein sogenanntes Bitboard [2] dargestellt. Das sind 64 Bit große Zahlen, die bestimmte Aspekte des Schachbretts speichern. Dies kann z.B. die Position von einer Typ von Spielfigur sein. Dabei entspricht ein Bit einer bestimmten Position auf dem Schachbrett. Mit Hilfe dieser Bitmuster kann man eine bestimmte Zugberechnung sehr schnell ausführen.

Mir kam es aber in meinem Schachcomputer nicht primär auf Geschwindigkeit an, sondern ihn so flexibel wie möglich zu machen. Deshalb hab ich mich für eine "Traditionelle" Datenstruktur entschieden. Es gibt insgesamt 32 Figuren auf dem Schachbrett auf beiden Seiten. Dabei kann jeder Bauer theoretisch zu einer Dame oder einer anderen höheren Figur durch Erreichen der hintersten Linie werden. Also hat man im Extremfall bis zu 9 Damen auf dem Spielfeld (siehe [5, Seite]). Deshalb ist eine sehr flexible Datenstruktur nötig. Ich speichere deshalb nicht nur die Position einer Figur (*figures*) ab, sondern auch die Art und Weise ihrer Bewegung (*MoveType*). Damit schnell bestimmt werden kann ob sich an einer bestimmten Position eine Figur befindet, wird zusätzlich noch ein Array verwaltet, das dem Schachbrett entspricht *chessfield*.

Zur schnelleren Berechnung der Bewertungsfunktion werden in einer zusätzlichen Datenstruktur Informationen gespeichert, zu welchen Feldern sich jede Figur bewegen bzw. geschlagen werden kann. Dies wird in *codefield* gemacht.

Wird nun ein Zug mit Hilfe der Methode *draw* durchgeführt werden die Datenstrukturen entsprechend geändert. Ursprünglich war auch geplant in *codefield* nur die Stellen zu ändern, die durch den Zug beeinflusst wurden. Dies hat sich aber als sehr aufwändig herausgestellt, weil man auch den Einfluss anderer Figuren neu berechnen muss, falls man eine Figur verschiebt oder schlägt. So ist einfacher die Datenstruktur einfach bei Bedarf komplett neu zu berechnen.

3.3 Spielregeln

Wie in Kapitel 2.1 schon beschrieben kann man die Schachregeln gut strukturieren. Diese Struktur wurde fast direkt in den C++ Code übernommen. Um dabei eine maximale Optimierung zu erreichen wurden Templatefunktionen verwendet. Wenn für eine bestimmte Figur Züge generiert werden, wird ihnen das Zugverhalten mit Hilfe von Templateparametern übergeben. So kann der Compiler während des Compilierens Vereinfachungen vornehmen.

Die Funktion *Field::makeMove* erzeugt die **Bewegungszüge**. Bei jedem Aufruf wird ihr über ein Templateparameter die Richtung des Zuges (dirx, diry), die aktuelle Seite (black), die auszuführende Operation (gt) und die maximale Zugweite (a) übergeben. Diese Informationen entsprechen denen, die in Kapitel 2.1 erarbeitet wurden. Die auszuführende Operation ist im Moment nur die Erzeugung der möglichen Züge.

Die Funktion *Field::makePosMove* erzeugt die **Springzüge**. Ihre Templateparameter sind: Sprungweite (*dirx, diry*), die Seite (*black*), die auszuführende Operation (*gt*) und die Art des Zuges (*movetype*). Mit movetype wird auch das spezielle Verhalten von Bauern realisiert.

Diese beiden Funktionen werden aus *Field::handleFigure* aufgerufen. In dieser Funktion wurden die Tabellen aus Kapitel 2.1 eingearbeitet. So wird nun jeder gültige Schachzug erstellt bis auf die Rochade und die Bauerntransformation.

Die Bauerntransformation wird in *Field::draw* erledigt. Und zwar wird jeder Bauer, der die hinterste Linie erreicht, automatisch in eine Dame umgewandelt.

Um die Rochade zu realisieren wird in *Field::allowSmallRochade* und *Field::allowBigRochade* gespeichert ob der jeweilige Spieler noch die Rochade durchführen kann. Schwarz kann dabei nur die kleine machen Weißnur die Große Rochade. Falls ein Spieler einen Zug macht mit dem König oder dem zugehörigen Turm so wird diese Variable auf *false* gesetzt. Eine Rochade ist dann nicht mehr möglich. Falls dies noch nicht geschehen ist wird bei der Zuggenerierung getestet ob die Felder zwischen König und Turm frei sind und die Felder, die den König angehen nicht bedroht sind. Wenn ja, wird die Rochade zu den gültigen Zügen hinzugefügt.

3.3.1 Schach-Test

In einem Schachcomputer ist es nicht nur nötig die möglichen Zügen der Figuren einzugeben, sondern auch globale Regeln zu beachten. Dies ist z.B. dass der Gegenspieler niemals einen Zug machen darf, der den König schlagen wurde. Deshalb muss man den König, falls er im Schach steht, aus ihm entfernen. Ebenso darf man keinen Zug machen wodurch der eigene König ins Schach kommen würde. Dies muss natürlich ebenso von einem Schachcomputer getestet werden.

Man kann diesen Test relativ einfach mit Hilfe der *codefield* Datenstruktur realisieren. Da in dieser auch drin steht welche Figur durch welche andere bedroht wird, braucht man nur zu überprüfen, ob der eigene König bedroht wird. Dann steht er im Schach. Dieser Test wird nach der Generierung jedes Zuges für die Seite gemacht, die gezogen hat. Falls der König dann im Schach steht, könnte er geschlagen werden. Deshalb wird der Zug dann verworfen. So wird garantiert, dass keine falschen Züge generiert werden.

3.3.2 Matt-Test

Dieser Test stellt fest, ob eine Seite gewonnen hat oder nicht. Es wird zuerst getestet, ob der König im Schach steht oder nicht. Falls ja, erzeugt man alle möglichen Züge aus dieser Position. Der eingebaute Schachtest verhindert, dass illegale Züge entstehen. So kann man ein Matt daran erkennen, dass der jeweilige Spieler von der Zuggenerierung keine Züge mehr bekommt. Dann hat der Gegenspieler gewonnen.

Auf Matt wird nur nach jedem Zug einmal getestet. Da von der Zuggenerierung keine illegalen Züge erzeugt werden, braucht man im eigentlichen Suchalgorithmus nicht auf Matt testen. Man kann es daran erkennen, dass ein Knoten keine Kinder mehr hat.

3.3.3 Patt-Test

Der Patt-Test ähnelt dem Matt-Test sehr. Der einzige Unterschied ist, dass er nur dann weitermacht, wenn der König nicht im Schach steht. Sind dann keine Züge mehr möglich, ist es ein Patt.

3.4 Suchalgorithmus

Die Klasse *AIPlayer* implementiert den Suchalgorithmus. Sie verwendet einen Iterative Deepening Alpha-Beta Algorithmus. Es wird eine Transposition Table, Move Ordering verwendet und Minimal Search Window auf der obersten Ebene.

Die Transposition Table ist in der Klasse *TranspositionTable* implementiert. Für einen schnellen Zugriff auf die gespeicherten Schachpositionen wird Hashing mit Verkettung verwendet. Die maximale Anzahl der gespeicherten Positionen wird zu Anfang festgelegt. Daraus wird ein verketteter Ring gemacht und es wird ein Zeiger auf das erste Element in *insertelement* gespeichert. Jedesmal, wenn man eine neue Position braucht, wird dieses Element neu in die Hashtabelle eingefügt. War das Element in *insertelement* schon vorher in der Hashtabelle enthalten, wird es vorher entfernt. So wird immer der älteste Eintrag überschrieben.

Der Hashwert für das Einfügen in die Transposition Table ist in der Eigenschaft "Field::hashcode" gespeichert. Dieser Wert wird einem nur beim Initialisieren des Schachbretts berechnet und dann sukzessive bei jedem Zug aktualisiert.

Als Move Ordering Strategie wird die statische Bewertungsfunktion verwendet. So muss man für meisten der Züge nur mit einem Minimal Search Window suchen.

Im eigentlichen Suchalgorithmus wird ein eigener MemoryManager verwendet um schnell Speicher für neue Züge zu bekommen. Dieser ist in der Klasse MemManager implementiert. Er verwaltet eine Menge von Schachfeldern, Zügen und ein Array zum sortieren für die Move Ordering Strategie.

3.5 Bewertungsfunktion

Die Berechnung der Bewertungsfunktion wird in Klassen realisiert, die von Evaluator abgeleitet sind. Dabei gibt es 2 verschiedene Implementierungen. Es ist eine herkömmliche Bewertungsfunktion die ohne den genetischen Algorithmus arbeitet. Sie befindet sich in der Klasse SimpleEval. Und eine komplexere, die durch den genetischen Algorithmus eingestellt werden muss. Sie befindet sich in der Klasse NormalEval.

Beide Klassen bewerten fast die gleichen Werte. Bei NormalEval kommt noch eine komplexere Bewertung der Bauern hinzu. Ebenfalls ist sie feiner einstellbar. Man kann z.B. bei der Mobilität jeder einzelnen Figur eine Gewichtung angegeben. Ebenso kann man zu jedem Figurenpaar einstellen, wieviel es bringt, wenn eine bestimmte Figur eine andere bedroht.

Diese Funktionen lassen sich in die Klasse AIPlayer einsetzen, in der sie zur Bewertung der Figuren eingesetzt werden.

Um die Geschwindigkeit bei der Bewertung des Schachfeldesfeldes zu erhöhen, werden bestimmte Bewertungen nur bei Bedarf aktualisiert oder sukzessive berechnet. Der Test wieviele freie Bauern und Doppelbauern sich auf dem Feld befindet wird nur gemacht, wenn ein Bauer verschoben oder geschlagen wurde. Und der Figurenwert und die Bauernposition werden nur einmal am Anfang berechnet. Danach werden nur noch die Änderungen aktualisiert.

4 Ergebnisse

4.1 Leistung

Der Schachcomputer berechnet durchschnittlich 26.000 Züge pro Sekunde und erreicht meistens eine Tiefe von 6 oder 7. Dies ist abhängig von der Anzahl der möglichen Züge in einer Position. Getestet wurde dies auf einem Intel Pentium M processor mit 1,5 Ghz. Der Testrechner hat 512 MB Hauptspeicher. Als Betriebsystem wurde SUSE Linux 10.2 verwendet.

Compiliert wurde das Programm mit dem GNU C++ Compiler. Als Optimierungsstufe wurde -O2 gewählt.

5 Fazit

Algorithmen sind Wissen, also Aussagen der Wissenschaft. Mit Hilfe dieser Aussagen habe ich versucht einen möglichst guten Schachcomputer zu konstruieren. Und weil dies so ist, kann man ihn immer wieder verbessern. Er ist deshalb kein Endprodukt. Besonders schade fand ich es, dass die Zeit nicht ausreichte um mit dem Genetischen Algorithmus die Bewertungsfunktion zu optimieren. Trotzdem bin ich mit dem Gesamtergebnis zufrieden. Mir ist es gelungen einen funktionierenden Schachcomputer zu erstellen. Es wurden moderne Techniken zur Optimierung verwendet.

Für mich ist das Projekt aber nicht mit dem Ende dieses Praktikums abgeschlossen. Es würde mich besonders reizen eine verteilte Version des Schachcomputers zu schreiben. Dieser könnte dann seine Rechenlast auf mehrere Computer verteilen. Dazu werden in [8] verschiedene Algorithmen beschrieben und verglichen.

Ebenso würde ich gerne die statische Bewertungsfunktion verbessern. Die Klasse NormalEval enthält sehr komplexe Auswertungsmechanismen. Durch Abstimmung der einzelnen Bewertungen könnte man sicherlich noch die Bewertungsfunktion verbessern und damit die Stärke des Schachcomputers erhöhen.

Für mich hat es sich auf jeden Fall gelohnt, denn durch die Entwicklung habe ich viel gelernt.

6 Literatur

Literatur

[1] Stuart J. Russell, Peter Norvig (2003). *Artificial Intelligence: A Modern Approach Second Edition* New Jersey, Pearson Education

[2] Heinz, Ernst A (2000). *Scalable Search in Computer Chess* Braunschweig / Wiesbaden, Friedr. Vieweg & Sohn Verlagsgesellschaft mbH

[3] Schöning, Uwe (2001). *Algorithmik* Heidelberg, Berlin: Spektrum, Akad. Verl.

[4] Ian Millington (2006). *Artificial Intelligence for Games* San Francisco Elsevier

[5] Euwe, Max (2003). *Schach von A-Z, 6.Auflage* Hollfeld Joachim Beyer Verlag

[6] Dr. Lazlo Orban (2007). *Schach für Anfänger, 3. Auflage* Baden-Baden, Humboldt Verlags GmbH

[7] Winston, Patrick Henry (1984). *Artificial Intelligence, Secound Edition* Addison Wesley

[8] Mark Gordon Brockington (1998) *Asynchronous Parallel Game-Tree Search* University of Alberta

[9] GNU Chess ?

[10] WxWidget Doc.