Die Programmierung eines Computers für das Schachspiel

Roman Hartmann rhartmann@bluewin.ch

27. Mai 2004

Maturaarbeit

Fach: Mathematik

Inhaltsverzeichnis

1 Einleitung

Das Schachspiel erfreut sich seit hunderten von Jahren grosser Beliebtheit und ist wohl neben go das meistverbreitete Brettspiel überhaupt. Das Duell zwischen Fisher und Spassky 1972 war ein vielbeachtetes Ereignis, das teilweise sogar im Fernsehen übertragen wurde. Interessant war dieser Anlass für den Grossteil des Publikums natürlich in erster Linie wegen des „Wettkampfes“ USA–UdSSR. Sozusagen ein Kampf der Systeme, der unblutig auf dem Brett ausgetragen wurde. Allerdings war es für normale Schachspieler kaum nachvollziehbar, was die weltbesten Spieler da genau auf dem Brett machten. Man war bei solchen Übertragungen immer auf die Analyse und Kommentare von Grossmeistern angewiesen, die einem die Feinheiten des Spiels erläuterten. Dies und der Umstand, dass sich die Spiele meist über Stunden hinzogen und dann oft mit einem Remis (Unentschieden) endeten, führte dazu, dass sich die Mehrheit des Fernsehpublikums auf Dauer verständlicherweise nicht so wirklich für Schach begeistern konnte. Das Ganze war für Nichteingeweihte einfach viel zu statisch und trocken und so wurde Schach medial bald wieder zurück in die Zeitungen verbannt.

Dies änderte sich, als IBM mitte der 90er Jahre den damaligen Schachweltmeister Garry Kasparov mit einem Computer herausforderte. Deep Blue (in Anlehnung an Big Blue für IBM), wie der Rechner von den Ingenieuren getauft wurde, war mit 200 speziellen Prozessoren bestückt, die fürs Schachspielen optimiert waren. Jeder der Prozessoren konnte 2 bis 3 Millionen Halbzüge pro Sekunde generieren. Trotz dieser gewaltigen, rohen Rechenkraft gewann Kasparov den ersten Wettstreit 1996 in Philadelphia klar 4–2. Für IBM hatte sich die Investition von etwa 20 Millionen Dollar, soviel hatte sich IBM die Entwicklung von Deep Blue kosten lassen, aber trotzdem gelohnt, denn dem Duell Mensch–Maschine war grosse Aufmerksamkeit beschieden und so war das Ganze trotz der Niederlage von Deep Blue ein riesiger Marketingerfolg für IBM. Aber auch das Schach erhielt generell wieder etwas mehr Aufmerksamkeit. Die zweite Auflage des Duells, 1997, bei der nun ein stärkerer Computer mit 512 Prozessoren[1] zum Einsatz kam, wurde erneut von vielen Menschen rund um den Globus interessiert verfolgt. Diesmal verlor Kasparov knapp gegen den Rechner mit 2.5–3.5. Der spätere Vorwurf Kasparovs[2], dass IBM zusätzlich die Hilfe von Grossmeistern in Anspruch nahm, die mit Deep Blue's Rechenkraft die aussichtsreichsten Züge untersuchten, konnte allerdings nie so ganz entkräftet werden.

Wie auch immer, das Thema Schach und insbesondere Computerschach erfreute sich nach dem Duell Deep Blue–Kasparov plötzlich grösserer Beliebtheit, da mittlerweile sowohl leistungsfähige Personalcomputer, wie auch starke Schachprogramme zu moderaten Preisen erhältlich waren, die durchaus bereits einen Grossmeister in Verlegenheit bringen konnten. Damit war es nun erstmals möglich, dass selbst durchschnittliche Spieler mit Computerhilfe Partien von Kasparov und Co analysieren konnten, ohne die Hilfe von starken Spielern bemühen zu müssen.

1.1 Kurze Geschichte des Schachspiels

Schach hat seinen Ursprung in Indien und erste Zeugnisse des Spiels datieren aus dem Jahr 500. Von Indien gelangte das Brettspiel nach Persien und mit den Mauren gelangte das Spiel im 10. Jahrhundert nach Spanien und damit auch nach Europa. Das Ziel des Schachspiels ist es, den gegnerischen König Matt zu setzen. „Shah mat“ bedeutet im

[1] Deep Blue II konnte rund 200 Millionen Stellungen (Halbzüge) pro Sekunde bewerten

[2] Kasparov gestand allerdings auch einen groben Fehler seinerseits ein. Zudem übersah er, dass eines seiner aufgegebenen Spiele durch Dauerschach in ein Remis zu retten gewesen wäre.

Arabisch-Persischen „Der König ist tot“. Die Form des Schachs, so wie wir es heute kennen, entstand im 15. Jahrhundert, vermutlich auch in Spanien. Die Gangart des Läufers, der bis dahin nur auf das zweite Feld in der Diagonale ziehen und schlagen konnte, wurde angepasst, und die Dame wurde zur mächtigsten Figur auf dem Brett, da sie nun sowohl Turm- als auch Läuferzüge ausführen konnte. Seither sind die Regeln des Spiels aber praktisch unverändert geblieben und die letzen Regelanpassungen erfolgten im 19. Jahrhundert

1.2 Kurze Geschichte des Computerschach

Die nachfolgende chronologische Auflistung von Ereignissen [vR] erhebt keinerlei Anspruch auf Vollständigkeit. Im Zeitraum von 1960 bis 1980 gab es eine Vielzahl verschiedener interessanter Entwicklungen im Computerschach, die es eigentlich ebenfalls verdienen würden, genannt zu werden. Da dies aus Platzgründen aber nicht möglich ist, werden hier nur ein paar, aus rein subjektiver Sicht natürlich, wichtige Marksteine erwähnt.

Der Wunsch nach intelligenten Maschinen oder Apparaturen, die Intelligenz mindestens vortäuschen, ist schon alt. So wurde 1769 von einem Baron von Kempelen am Hofe der Kaiserin Maria Theresia ein Automat vorgeführt, der Schach spielen konnte oder zumindest den Anschein erweckte, dies zu tun. Der Automat bestand aus einer menschengrossen Puppe in türkischer Tracht (alles türkische war damals gerade en vogue). Diese Puppe sass an einem grossen Schachbrett und konnte die Figuren darauf bewegen, während aus dem innern des Apparates das Quietschen und Rattern von Zahnrädern zu hören war. Der Schachtürke, wie er bald getauft wurde, war eine ziemliche Sensation seiner Zeit. Viele Schachspieler, darunter auch Staatsmänner wie Napoleon, spielten gegen den Automaten. Die Mehrzahl der Partien entschied der spielstarke Schachtürke für sich. Der Schachautomat war allerdings in erster Linie eine perfekte Täuschung, verbarg sich doch im innern des Automaten ein hervorragender Schachspieler, der in Wirklichkeit die Züge ausführte. Erstaunlicherweise flog der Betrug aber nie so richtig auf, obwohl der Automat von tausenden Menschen in ganz Europa bestaunt wurde und einige wenige helle Köpfe den Schwindel[3] auch erkannten aber kaum Gehör fanden, da die Menschen einfach glauben wollten, der Automat wäre „echt“.

Der erste erfolgreiche Versuch einer Maschine das Schachspiel beizubringen, wenn auch nur ein elementares Endspiel, gelang 1912 dem Spanier Torres Y Quevedo. Sein auf mechanische Weise funktionierendes Gerät konnte mit weissem König und Turm den gegnerischen schwarzen König Matt setzen, dies allerdings nicht immer auf kürzestem Weg.

Nach dem Ende des 2. Weltkrieges beschäftigte sich unter anderem auch der englische Mathematiker Alan Turing mit der Theorie der Schachprogrammierung. Mangels Computer konnte er sein Programm aber nur aufs Papier bringen. Dieses „Schachprogramm“ enthielt aber bereits eine Vielzahl von Ansätzen, die teilweise auch in heutigen Schachprogrammen noch Verwendung finden. Die Züge mussten von Turing noch mühsam und zeitaufwendig im Kopf berechnet werden und eine Partie, die 1952 zwischen Turing und einem Hobbyspieler bestritten wurde, dauerte dann auch nur 22 Züge, ehe Turings „Papiercomputer“ Matt gesetzt wurde.

Als weiterer Pionier des Computerschachs gilt Claude E. Shannon. Sein 1950 im Philosophical Magazin veröffentlichter Vortrag [Sha50] „Programming a Computer for Play-

[3] Der Ausdruck „türken“, für jemanden hinters Licht führen, kommt wahrscheinlich von dieser Begebenheit.

ing Chess“ war wegweisend für die weitere Entwicklung des Computerschachs. Shannon glaubte, dass die bei der Schachprogrammierung gewonnenen Erkenntnisse und Problemlösungsstrategien sich auch auf andere Bereiche der Informatik anwenden lassen würden. Schach bot sich als Betätigungsfeld für Informatikpioniere ja auch geradezu an, da die Regeln eindeutig sind und das Brett mit den 64 Feldern und den 32 Figuren den Programmierer scheinbar nicht vor unüberwindbare Hürden stellt.

1951 gelang es Dietrich Günter Prinz an der Universität in Manchester ein einfaches, zweizügiges Mattproblem mit einem Computer zu lösen. Der Rechner brauchte gut 15 Minuten, um den für menschliche Spieler sofort ersichtlichen Gewinnzug zu finden.

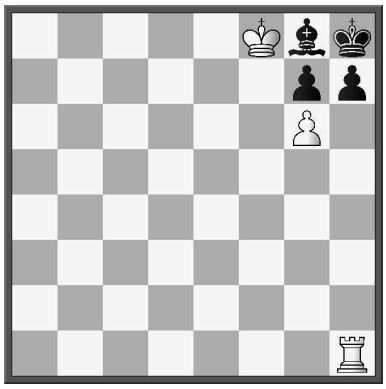

Abbildung 1: Matt in 2 eingeleitet mit Th6 (1. Th6 g7xh6 2. g7#)

Ein anderes Kapitel war der Univac MANIAC I. Dieser programmierbare Rechner, der in den USA in Los Alamos hauptsächlich für die Kernforschung[4] genutzt wurde, war einer der leistungsfähigsten Rechner seiner Zeit. Der MANIAC I, ausgestattet mit 18'000 Elektronenröhren, konnte 10'000 Operationen pro Sekunde ausführen und wurde 1956 von einem Forscherteam so programmiert, dass er auf einem reduzierten 6x6 Brett Schach spielen konnte. Zur weiteren Vereinfachung waren auch die Doppelschritte des Bauern und Rochaden nicht in das Programm implementiert und die Läufer gänzlich weggelassen worden. Trotz der Simplifizierung brauchte der Rechner ganze 12 Minuten, um nur einen einzigen Zug zu berechnen. In dieser Zeit rechnete der Rechner gerade mal vier Halbzüge tief. Da Rechenzeit damals noch ein rares Gut war, spielte der Rechner dann auch nur drei Partien.

1980 wurde von Ken Thompson[5] ein Computer fertiggestellt, der nur für das Schachspiel gebaut wurde. Als erster benutzte Thompson spezielle Chips, die für die Generierung und Bewertung von Schachzügen optimiert waren. Thompson rechnete während der Planungsphase damit, dass dieser Rechner nach seiner Fertigstellung über eine Million Züge pro Sekunde bewerten würde. Obwohl der Rechner schlussendlich „nur“ etwa 120'000 Züge pro Sekunde bewerten konnte, war er doch der weitaus stärkste Schachcomputer seiner Zeit und das Konzept, spezielle Chips für die Generierung der Züge und deren Bewertung zu verwenden, wurde einige Jahre später auch erfolgreich bei Deep Blue verwendet.

[4]Die ersten Programme, die auf dem MANIAC liefen untersuchten die Realisierbarkeit einer Wasserstoffbome

[5]Ken Thompson ist auch einer der Väter von UNIX und der Programmiersprache C

1.3 Wie ein Mensch (Grossmeister) Schach spielt/Mustererkennung gegen rohe Rechenkraft

Bevor wir uns dem eigentlichen Thema, der Funktionsweise eines Schachprogramms, zuwenden, ein kurzer Exkurs darüber, was man über das menschliche Schachspiel weiss.

Lange glaubte man, dass besonders talentierte Schachspieler über eine überdurchschnittlich schnelle Kombinationsgabe verfügen müssten, um mehr Züge als der Gegner vorausplanen zu können. Eine Studie [CS], die mit Grossmeistern und Hobbyspielern durchgeführt wurde und 1970 publiziert wurde, zeigte dann aber Erstaunliches. Die Teilnehmer der Studie mussten Brettstellungen, die ihnen für etwa zehn Sekunden gezeigt wurden, aus dem Gedächtnis rekonstruieren. Während die Grossmeister in der Lage waren, gut 90% der etwa 25 Figuren auf dem Brett wieder korrekt zu platzieren, konnten die Hobbyspieler im Durchschnitt nur etwa 30% der Steine wieder auf das richtige Feld stellen. Wenn nun aber den Studienteilnehmern zufällige Brettstellungen, die so von erfahrenen Schachspielern niemals gespielt worden wären, vorgesetzt wurden, dann konnten auch die Grossmeister die Stellung nicht mehr korrekt aufbauen, und es zeigten sich plötzlich kaum noch Leistungsunterschiede. Sowohl Grossmeister wie Hobbyspieler konnten nur noch rund 20% der Figuren richtig platzieren.

Es scheint also, dass Grossmeister in erster Linie in der Lage sind, Muster zu erkennen, die sie schon öfters auf dem Brett angetroffen haben und daraus auch die Entschlüsse für ihr weiteres Vorgehen im Spiel fassen.

Stark vereinfacht ausgedrückt, erkennt der erfahrene Schachspieler den Erfolg versprechenden Zug so, wie wir ohne Mühe ein uns vertrautes Gesicht in einer Menschenmenge entdecken.

Nach einer groben Schätzung dieser Studie müssen im Gedächtnis eines Schachspielers mindestens 50'000 Muster (Brettstellungen) „abgespeichert" sein, damit er (oder sie) die Grossmeisterreife erreicht. Andere Annahmen bewegen sich zwischen 100'000 und 300'000 abgespeicherten Mustern. Ein gewisses Talent, gepaart mit jahrelanger harter Arbeit, ist natürlich unabdingbare Voraussetzung, um dies zu erreichen. Schach scheint also mehr ein Gedächtnisspiel als ein Kombinationsspiel zu sein, zumindest für die Profis.

1.4 Motivation für die Schachprogrammierung

Alan Turing definierte eine Maschine als intelligent[6], wenn beim Kommunizieren mit ihr, nicht mit Sicherheit gesagt werden kann, ob es sich um einen Menschen oder eine Maschine handelt. Diese Aussage kann natürlich dahingehend erweitert werden, dass ein Computer als intelligent zu betrachten ist, wenn beim Schachspiel mit ihm nicht erkennbar ist, ob da nun ein Mensch oder ein Computer die Züge wählt. Der Grund, dass sich so viele Forscher mit der Schachprogrammierung befassten, hängt auch damit zusammen, dass man glaubte, mit der Schachprogrammierung auch etwas über die menschliche Intelligenz und Lernprozesse herauszufinden, indem man diese Prozesse in einer Maschine nachzuvollziehen versuchte.

Es gab in den Anfängen der Schachprogrammierung zwei gänzlich unterschiedliche Ansätze, dem Computer das Schachspiel beizubringen. Das eine Lager versuchte dem Computer möglichst viel Schachwissen und menschliche Denkmuster „einzuimpfen" und damit das menschliche Schachspiel möglichst zu kopieren. Man glaubte, die Computer so programmieren zu können, dass sie zudem aus jeder gespielten Partie lernen würden. Die Programme sollten damit von Spiel zu Spiel stärker werden. Die zweite Gruppe setzte

[6]Das Ganze ist auch als Turing Test bekannt

auf reine Rechengeschwindigkeit und ein absolutes Minimum an Schachwissen. Obwohl es gerade auf der Seite der „intelligenten“ Schachprogramme sehr interessante und viel versprechende Versuche gab, setzten sich die simplen Rechenknechte in Sachen Spielstärke klar durch. Es zeigte sich, dass sich menschliche Problemlösungsstrategien nicht so einfach auf einen Computer übertragen liessen. Die menschlichen Denkprozesse sind offenbar zu komplex, als dass sie sich so einfach auf einem Computer nachformen liessen. Dass die Schachprogrammierung so einfach nicht sein kann, zeigt auch das Beispiel des Schachweltmeisters Botwinnik [Bot82], der über 30 Jahre lang mehr oder weniger erfolglos an einem intelligenten Schachprogramm (Pionier) arbeitete. Allerdings muss auch erwähnt werden, dass die Realisierung der intelligenten Programme auch meist schon and dem Umstand scheiterte, dass die Computer in den 50er- und 60er-Jahren einfach noch zu schwach waren, um diese Ideen auch umsetzen zu können.

In dieser Maturaarbeit wird ausschliesslich der Aufbau und die Funktionsweise des zweiten Typus von Schachprogrammen erläutert. Programme also, die mit relativ wenig Schachwissen ausgestattet sind, dafür aber systematisch eine Vielzahl von Zügen in kurzer Zeit untersuchen können.

Hier ist das Layout eines solchen Schachprogramms:

1. Zuggenerator: erzeugt alle legalen Züge
2. Suche: führt alle eigenen und gegnerischen Züge aus
3. Bewertung: Bewertet die Endstellung und trifft die Entscheidung, welche dieser Zugfolgen für das Programm am aussichtsreichsten sind

gehen wir nun aber etwas mehr ins Detail:

2 Zuggenerierung

Eines der Probleme das als erstes gelöst werden muss, ist es, dem Computer die Stellung der Figuren auf dem Brett beizubringen. Am einfachsten geschieht dies natürlich durch ein 8x8 Feld. Damit ergeben sich aber Probleme bei der Zuggenerierung, denn ungültige Springerzüge, die den Arraybereich überschreiten, müssen rechtzeitig erkannt und ihre Ausführung verhindert werden. Dies macht die Zuggenerierung unnötig aufwendig.

2.1 10x12 Brettrepräsentation

Dieses knifflige Problem wurde von Claude Shannon bereits in den 40er Jahren dadurch gelöst, dass er das 8x8 Feld auf ein 10x12 Feld[7] erweiterte. Damit wird sichergestellt, dass auch ungültige Züge über das 8x8 Feld hinaus, nicht zu einer Bereichsüberschreitung führen. Da bei Computern eindimensionale Arrays (Felder) üblicher sind, wird das 10x12 Feld meist als Array von 0-119 dargestellt. Auf das Feld A1 kann somit beispielsweise mit board[21] zugegriffen werden. Die Figuren werden vorzugsweise über Integerwerte identifiziert, 1 für einen weissen Bauern, 2 für einen weissen Springer, 3 für einen weissen Läufer, 4 für einen Turm, 5 für die Dame, 6 für den König. Die schwarzen Figuren werden analog mit den gleichen Werten aber mit negativem Vorzeichen versehen. Die Randfelder werden Beispielsweise mit dem Wert 100 gekennzeichnet und die leeren Felder mit dem Wert 0.

[7] Grafiken qualitativ übernommen aus [DM97]

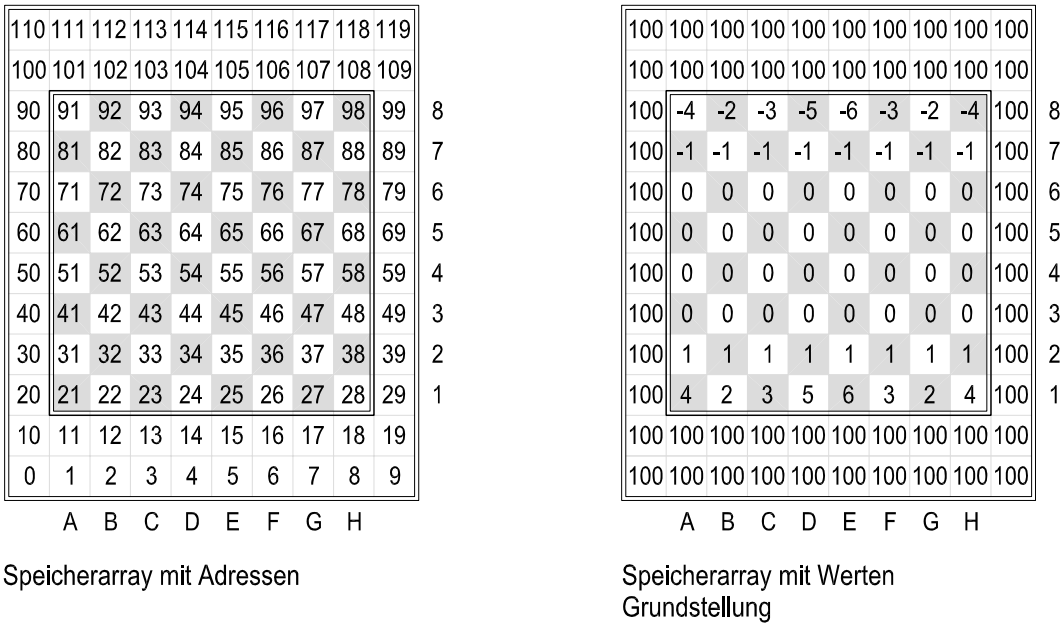

Abbildung 2: 10x12 Feld mit Adressen und zugewiesenen Werten in Grundstellung

Eine Routine geht jetzt systematisch durch den gesamten Array, prüft, ob dort bereits eine Figur steht und wohin diese allenfalls ziehen, oder welche gegnerische Figur sie schlagen kann und speichert die Züge anschliessend in einer Liste.

Wenn nun beispielsweise der weisse Springer im 10x12 Feld von B1 auf A3 springt, wird das im Rechner als board[22+19] = 2 umgesetzt (wobei board[22] nach erfolgtem Zug natürlich auf null gesetzt werden muss). Unter Fachleuten wird dabei die Zahl 19 als Offset bezeichnet. Für den Springer weitere mögliche Offsets sind 21, 8, 12, -21, 19, -19, -8, -12. Die Züge für die anderen Figuren werden analog erzeugt. Gültige Offsets für den König sind 1, -1, 10, -10, 9, -9, 11, -11. Diese Offsets haben natürlich nur für das 10x12 Feld Gültigkeit. Für die weissen Figuren muss bei der Zuggenerierung lediglich geprüft werden, ob der gespeicherte Wert des Zielfeldes kleiner oder gleich Null ist. Falls das Zielfeld den Wert 0 hat, dann ist es leer und falls es kleiner als Null ist, dann beherbergt das Feld eine schwarze Figur, die geschlagen werden kann. Für die schwarzen Figuren muss bei der Zuggenerierung geprüft werden, ob das Zielfeld einen Wert grösser oder gleich Null aufweist und zusätzlich sichergestellt werden, dass das Zielfeld nicht auf dem Rand liegt.

Da das Schachspiel ein symmetrisches Spiel ist, können für Weiss und Schwarz die gleichen Routinen für die Zuggenerierung verwendet werden, abgesehen von den Bauern, die ja in jeweils entgegengesetze Richtung marschieren.

Nachfolgend der Quellcodefragmente[8] für die Generierung der Springerzüge von Weiss. Eine kurze Erklärung zur C-Syntax findet sich im Kapitel 3.1.2.

```
/* Springer Offsets fuer 12x10 Feld */
const int mv_s[8] = { 19, -19, 8, -8, 12, -12, 21, -21 };
        ...
        /* Schleife durch gesamtes Brett */
        for (i = 98; i > 20; i--)
        {
```

[8]entnommen aus ROCE „Roman's Own Chess Engine“ meinem eigenen Schachprogramm

```
while (p->board[i] == LEER || p->board[i] == RAND
       || p->board[i] < 0 && i > 20)
  i--;

/* Routine fuer weissen Springer */
if (p->board[i] == Springer_weiss)
  {
    for (j = 0; j < 8; j++)
      {
        /* wenn das Brett leer ist oder eine */
        /* schwarze Figur auf dem Feld steht */
        /* kann auf das Feld gezogen werden */
        if (p->board[i + mv_s[j]] < 0
            || p->board[i + mv_s[j]] == LEER)
          {
        /* t_mv(..) prueft ob der Zug legal ist */
        /* der eigene Koenig darf nach dem Zug */
        /* nicht im Schach stehen */
            if (t_mv (p, i, i + mv_s[j]) == TRUE)
              {
                list->von = i;
                list->nach = i + mv_s[j];
                list->umwandlung = FALSE;
                list->rochade = FALSE;
                list->value = p->board[i + mv_s[j]];
                list++;
              }
          }
      }
  }
.....
```

Jetzt brauchen wir nur noch Routinen für en-passante Züge, die Rochade und die Umwandlungen zu implementieren und damit haben wir bereits einen funktionsfähigen Zuggenerator, der uns für die jeweilige Stellung sämtliche (pseudolegalen) Züge erzeugen kann. Unsere Routine muss jetzt zusätzlich natürlich auch noch sicherstellen, dass unser König bei keinem dieser Züge ins Schach gestellt wird oder aber nach einem gegnerischen Zug im Schach belassen wird. Ein in C oder Pascal (zwei gängige Programmiersprachen) geschriebener Zuggenerator umfasst rund 200-300 Zeilen Code.

Allerdings muss hier doch noch erwähnt werden, dass die Mehrheit der modernen Schachprogramme kaum mehr 10x12 Arrays für die Brettpräsentation und Zuggenerierung verwendet. Die modernen Programme verwenden heute zunehmend Bitboards für die interne Darstellung des Brettes.

2.2 Bitboards

Als Bitboard [Zib] wird dabei eine 64-Bit Zahl bezeichnet. Diese Zahl spielt nicht nur in der Computerwelt eine wichtige Rolle, sondern auch in der Schachwelt. Damit ist es nämlich möglich, mit einer einzigen Zahl das gesamte Brett abzudecken. Der Vorteil in der Verwendung von Bitboards liegt daran, dass Bitboards weitaus weniger Verwaltungsaufwand benötigen als Arrays und dies resultiert dann auch in einem schnelleren Programm.

Das gesamte Brett mit den Positionen aller schwarzen Figuren kann somit mit einer einzigen Zahl dargestellt werden:

```
Binär: 000000000 000000000 000000000 000000000 000000000 000000000 11111111 11111111
```

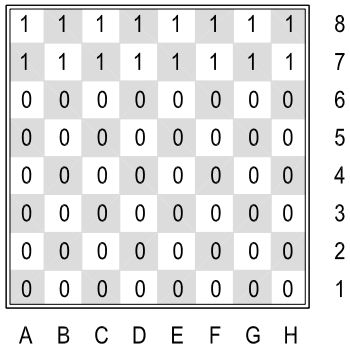

Abbildung 3: Schwarze Figuren in Grundstellung

```
Hexadezimal: 00 00 00 00 00 00 FF FF
Dezimal: 65535
```

2.2.1 Verwendung von Bitboards

Der grösste Vorteil von Bitboards liegt aber darin, dass die Bewertungsfunktionen nur noch ein Minimum an Operationen erfordern. Um zum Beispiel in Abbildung 4 die von Schwarz angegriffenen und von Weiss nicht verteidigten Figuren zu bestimmen, braucht es lediglich zwei verschiedene Funktionen und drei Bitboards.

Verwendete Bitboards:

- B1: Alle weissen Figuren auf dem Brett
- B2: Alle von weiss angegriffenen Felder
- B3: Alle von schwarz angegriffenen Felder

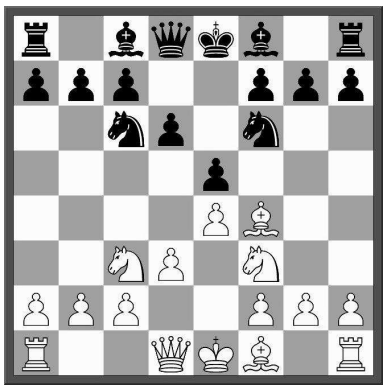

Abbildung 4:

Vorgehen:

Zuerst werden alle von Schwarz angegriffenen Figuren ermittelt und in der 64-Bit Variable B_1S gespeichert. Anschliessend werden mithilfe des Bitwise-COMPLEMENT die von Weiss nicht verteidigten Felder bestimmt und in Bitboard B_2S gespeichert.

Bitwise-AND von B_1S und B_2S ergibt dann die von Schwarz angegriffenen und von Weiss nicht verteidigten Figuren. In diesem Fall ist dies der Läufer auf F4. Der ganze Ablauf ist in Abbildung 5 detailliert dargestellt.

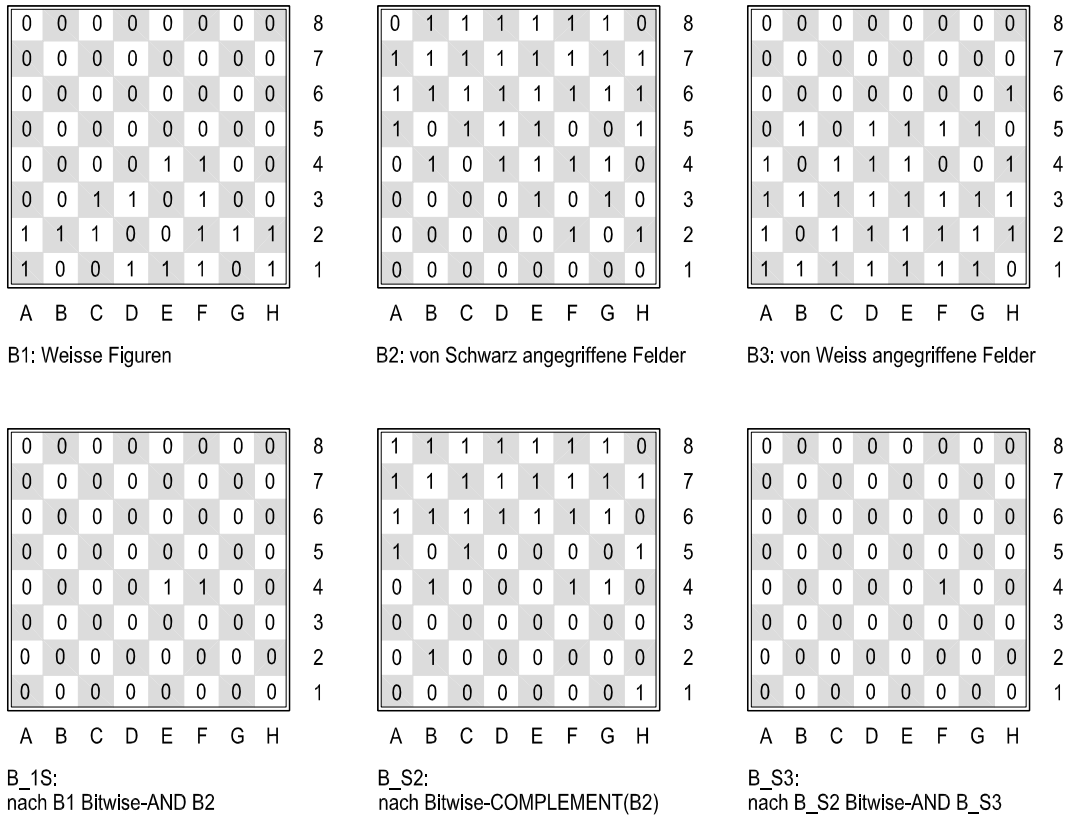

Abbildung 5: Bitboardverfahren

Mit lediglich drei Operationen, 2 mal Bitwise-AND und einmal Bitwise-COMPLEMENT, können wir hier also erkennen, welche weissen Figuren von Schwarz angegriffen werden und von Weiss nicht gedeckt sind. Wir sehen, dass der weisse Läufer auf F4 in Gefahr ist, durch Schwarz geschlagen zu werden, ohne dass auf dieses Feld zurückgeschlagen werden kann. Momentan (2004) hat die ganze Bitboardgeschichte nur einen kleinen Schönheitsfehler, der darin liegt, dass eine Vielzahl von Computern noch mit 32-Bit Prozessoren und nicht mit 64-Bit Prozessoren ausgestattet sind. Da die 64-Bit Werte vom Prozessor auf 32-Bit umgesetzt werden müssen, kann der Vorteil der Bitboards natürlich nicht voll ausgenützt werden. Trotzdem basieren einige der aktuellen und starken Schachprogramme bereits auf Bitboards [Hya04].

3 Die Suche

Wir haben nun einen funktionsfähigen Zuggenerator, der alle legalen Züge berechnet und in einer Liste abspeichert. Es geht nun darum, den aussichtsreichsten Zug, beziehungsweise die beste Zugfolge zu ermitteln. Dazu werden die legalen Züge vom Computer ausgeführt und die resultierenden Endstellungen mit einer Bewertungsfunktion, die einen Integerwert zurückliefert, bewertet. Die beiden gebräuchlichsten und meistverbreiteten Verfahren dazu sind das Mini-Max- und das Alpha-Beta-Verfahren.

3.1 Mini-Max

Mit der Mini-Max-Routine werden einfach sämtliche möglichen Züge und Gegenzüge bis zu einer festen Tiefe berechnet. Häufig wird allerding nicht mit einer festen Tiefe sondern mit einer festen Suchzeit gerechnet, da sich bei Endspielen mit wenigen Figuren auf dem Brett natürlich in kurzer Zeit grössere Suchtiefen erreichen lassen. Das sogenannte Mini-Max Prinzip geht dabei immer vom Grundsatz aus, dass auch der Gegner seinen besten Zug ausführen wird. Weiss versucht also den Zug, bzw. diese Zugfolgen auszuführen, die den maximal garantierten Wert für Weiss liefern, während Schwarz die Züge auswählt, die diesen Wert minimieren. Deshalb auch der Name „Mini-Max"-Verfahren. Wenn nun Weiss am Zug ist, erzeugt der Zuggenerator alle möglichen Züge für Weiss und führt diese der Reihe nach aus. Auf jeden dieser Züge folgt eine Anzahl schwarzer Gegenzüge, die wiederum von Weiss beantwortet werden. Bei einer vorgegebenen Länge, der so genannten Suchtiefe, wird dieser Prozess abgebrochen. Die resultierende Endstellung wird anschliessend von einer Bewertungsfunktion mit einer Zahl bewertet. Falls verschiedene Zugfolgen die gleiche Bewertung aufweisen entscheidet meist ein Zufallsgenerator, welche Variante ausgewählt wird oder aber es wird weitergerechnet.

Abbildung 6 zeigt einen Spielbaum aus einem Schachspiel. Um die Übersichtlichkeit zu wahren, sind dabei jeweils nur zwei Halbzüge[9] pro Seite aufgeführt. In Wirklichkeit liegt der Verzweigungsfaktor beim Schachspiel im Schnitt zwischen 30 und 40. Weisse Kreise bedeuten Weiss am Zug, die schraffierten Kreise bedeuten Zugrecht für Schwarz. Die Zahlen unter 4, 5, 7 ... sind das Resultat einer Bewertungsfunktion. Positive Zahlen ergeben einen Vorteil für Weiss, während negative Zahlen einen Vorteil für Schwarz bedeuten. Weiss ist am Zug und führt den Zug aus, der die höchste Endbewertung ergibt. In Stellung 3 wäre dies dxe5 und in Stellung 13 Kxf2. Schwarz am Zug agiert gleichermassen und wählt den Zug, der die tiefste Bewertung aufweist. In Stellung 2 wäre dies Dd5 und in Stellung 9 Tg5.

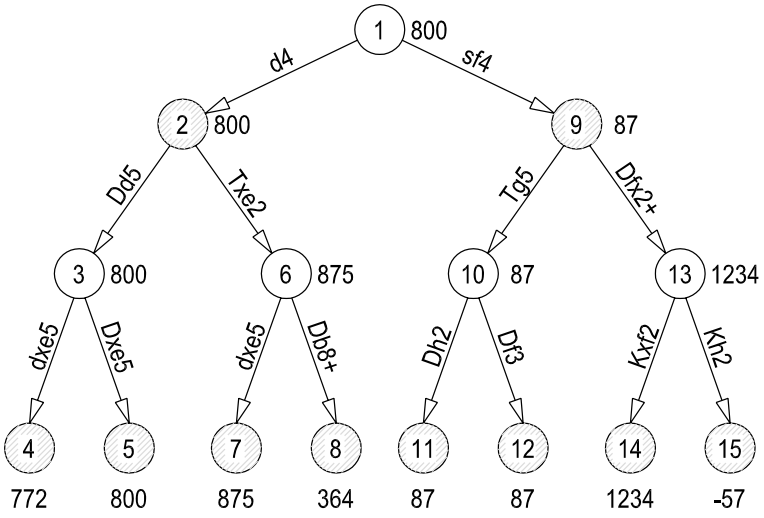

Abbildung 6: Minimax Spielbaum. Sämtliche Zweige werden untersucht.

Anschliessend folgt die vereinfachte und kommentierte Mini-Max Funktion aus einem Schachprogramm in C-Notation. Die Funktionsweise der Mini-Max Routine ist zwar mit Abbildung 6 ja bereits hinlänglich bekannt, aber der Programmausschnitt ist doch sehenswert. Die untenstehende Mini-Max-Funktion ist rekursiv (ruft sich also selbst wieder

[9] 1. e4, e5 wird als ein Zug bezeichnet während 1. e4 ein Halbzug ist

auf) und nimmt die Brettstellung und Suchtiefe als Argument und liefert den Wert für den besten Zug für Weiss oder Schwarz an die aufrufende Instanz zurück.

3.1.1 Mini-Max-Algorithmus / in C-Notation

(Entnommen und stark vereinfacht von Minimax.c von Dr. C. Donninger)

```
int minimax(struct position *p, int depth) {

    struct move list[MAXMOVES]; /* Datenstruktur: speichert Zuege */
    int i, m, t, score;

    if(depth == 0)              /* Horizont (Suchtiefe) erreicht */
        return evaluate(p);     /* Bewertungsfunktion wird aufgerufen */

    if(p->color == WHITE)
        t = -INFINITY;
    else
        t = INFINITY;

    m = makemovelist(p, list);  /* w: Anzahl legaler Zuege, list */
                                /* speichert die Zuege */

    if(m == 0)                  /* keine Zuege in dieser Pos. */
        return checkfordraw(p); /* Remis weil Patt */

    for(i = 0; i < m; i++) {    /* alle Zuege werden der Reihe nach */
        domove(list[i], p);     /* ausgefuehrt */
        score = minimax(p, depth-1);    /* Rekursiver Aufruf von MiniMax */
        undomove(p);            /* Zug wird rueckgaengig gemacht */

        if(p->color == WHITE)
                t = max(score, t);
        else
                t = min(score, t);
    }

    return t;                   /* Rueckgabe bew. des besten Zuges */
}
```

3.1.2 Einige Erläuterungen zur C-Notation

Programmierkenntnisse sind eigentlich keine nötig, um die hier aufgelisteten Algorithmen grundsätzlich zu verstehen. Da die C-Notation aber einige Besonderheiten aufweist, hier doch ein paar Erläuterungen zur Programmiersprache C:

```
int m;                  /* Eine Variable m wird erstellt, m */
                        /* speichert Integerwerte. */
                        /* Befehle werden immer mit einem Semikolon */
                        /* ; abgeschlossen */
m = 10;                 /* m erhaelt den Integerwert 10 */
                        /* zugewiesen. */
#define WHITE 1         /* Konstante WHITE mit Wert 1 */
for(m = 1, n = 0; m <= 100; m++)
        n = (m+n);      /* Schleife mit Variable m, summiert die */
                        /* Zahlen von 1 bis 100 und speichert */
                        /* diese in der Integervariablen n */
m++;                    /* entspricht m=(m+1), m wird somit um eins */
```

```
                        /* erhoeht */
int add(int a, int b)   /* Funktion, die zwei Integerwerte als */
        return(a+b);    /* Argumente nimmt und deren Summe */
                        /* zurueckgibt. */
if(n == m)              /* Bedingung, man beachte den Unterschied */
        return true;    /* zwischen = und == */
else                    /* wenn m gleich n ist wird true */
        return false;   /* zurueckgegeben, andernfalls false */

struct position {               /* erzeugt eine Datenstruktur Position,*/
        int color;              /* die Auskunft gibt ueber Position */
        int board[119];         /* der Figuren, Zugrecht, Rochade- */
        int rochade_l_weiss;    /* moeglichkeit und 50 Zug-Regel */
        int rochade_l_schwarz;
        int rochade_k_weiss;
        int rochade_k_schwarz
        int f_moves;
};


struct position p[100];         /* erzeugt 100 Strukturen vom Typ */
                                /* position */
p[0].color = WHITE;             /* p[0].color erhaelt einen Wert */
                                /* zugewiesen. WHITE muss dabei */
                                /* ein Integerwert sein */
                                /* Auf die verschiedenen Strukturen */
                                /* wird mit p[0-99].Variable zugegr. */
/* Dies ist ein Kommentar */    /* Hilft Menschen den Quelltext */
                                /* besser zu verstehen */
                                /* was zwischen /* */ steht wird vom */
                                /* Compiler bei der Generierung des */
                                /* Programmes ignoriert */
```

Die ersten funktionsfähigen Schachprogramme funktionierten praktisch ausnahmslos nach dem Mini-Max Prinzip. Die Spielstärke war aber oftmals nicht gerade berauschend, denn das Mini-Max Verfahren birgt ein Problem. Ausgehend von einer durchschnittlichen Schachstellung gibt es rund 36 Zugmöglichkeiten, die wiederum von rund 36 Möglichkeiten beantwortet werden können. Dieses Verfahren ist also nicht sehr effizient, da nach 5 Halbzügen bereits über 60 Millionen (36^6) Möglichkeiten geprüft werden müssen. Starke Schachspieler blicken da aber erstaunlicherweise immer noch durch (Da sie nur die „relevanten“ Züge berücksichtigen) und lassen sich davon noch nicht wirklich beeindrucken. Die ersten Schachprogramme, die auf dem Mini-Max-Verfahren basierten, waren dann auch noch chancenlos gegen starke menschliche Spieler.

Nach Berechnungen von Ken Thompson müssen die Computer etwa 4 Halbzüge tief rechnen, um ein Elo-Rating[10] von 1'200 Punkten zu erreichen. Bei jedem Halbzug, der nun tiefer gerechnet wird, ergibt sich ein beinahe linearer Spielstärkezuwachs von etwa 200 Punkten. Wenn nun ein Computer 5 Halbzüge tief rechnet, ergibt sich somit lediglich eine Spielstärke von etwa 1'400 Elo Punkten. Die stärksten Spieler der Welt haben aber Spielstärken von über 2'600 Elo Punkten. Kasparov weist sogar über 2'800 Punkte auf. Mit dem Mini-Max Verfahren kann diese Spielstärke also offenbar (noch) nicht erreicht werden.

[10]Mit der Elozahl wird die Spielstärke eines Spielers bezeichnet. Die Elo-Zahl ist keine unveränderliche Grösse und wird aus dem Verhältnis von Gewinnpartien und Verlustpartien unter der Berücksichtigung der Elo-Zahl der Gegner berechnet.

Das Problem ist offenbar das mangelnde Schachwissen, das den Rechner zwingt, alle möglichen Züge zu berechnen. Der Grossteil der Rechenzeit wird somit mit dem Berechnen von komplett sinnlosen Zugfolgen verschwendet. Wie kann dieses Problem umgangen werden?

3.2 Nega-Max

Hier handelt es sich eigentlich auch um Mini-Max aber in einer etwas vereinfachten Form. Bei Mini-Max versucht Weiss zu maximieren, während Schwarz zu minimieren versucht, was dazu führt, dass der Quellcode etwas umfangreich wird. Beim Nega-Max-Verfahren maximieren nun einfach beide Seiten, allerdings muss die Bewertungsfunktion nun, abhängig von der aufrufenden Seite, entweder positive oder negative Werte an die Suche zurückgeben.

```
int negamax(struct position *p, int tiefe)
{
    struct move list[MAXMOVES]; /* in list werden die Zuege gesp. */
    int m, i;
    int best = -INFINITY;       /* Rueckgabewert */

    m= makemovelist(p, list);   /* generiert legale Zuege */

    if (tiefe == 0)             /* Suchtiefe erreicht */
        return evaluate(p);     /* Endstellung wird bewertet */

    for(i = 0; i < m; i++) {    /* saemtliche legalen Zuege */
        domove(list[i], p);     /* werden ausgefuehrt */
        val = -negamax(p, tiefe - 1); /* und bewertet */
        undomove(p);            /* Zug wird rueckgaengig */
                                /* gemacht */
        if (val > best)         /* Endstellung besser ? */
            best = val;
    }
    return best;                /* Rueckgabewert bewertung */
}                               /* beste Endstellung */
```

3.3 Alpha-Beta

Da das Mini-Max Verfahren praktisch unüberwindbare Grenzen hat, die auch durch immer schneller werdende Rechner nur allmählich beseitigt werden, haben Programmierer das Mini-Max Verfahren zu verbessern versucht.

Eine Verbesserung des Mini-Max Algorithmus ist das Alpha-Beta Verfahren[11], das 1958 entdeckt wurde. Das Alpha-Beta Verfahren verfolgt nicht einfach alle möglichen Zweige im Spielbaum, sondern schneidet die völlig aussichtslosen Züge, die der Gegner sowieso niemals ausführen wird, frühzeitig ab. Dadurch muss eine Unmenge von Zügen gar nicht erst berechnet werden und bei gleicher Rechengeschwindigkeit verdoppelt sich damit die Suchtiefe. Auch hier wird natürlich davon ausgegangen, dass der Gegner den für ihn jeweils bestmöglichen Zug ausführen wird.

Abbildung 7 zeigt einen Ausschnitt aus einem Spielbaum. In Stellung 1 ist die Bewertung bereits bekannt. Der garantierte Wert für Weiss beträgt 800 und ist für Weiss die untere Schranke, die so genannte Alpha-Schranke. Für Schwarz bedeutet dieser Wert die obere Grenze, die Beta-Schranke. Bei Stellung 6 wird vom Programm der erste Zug

[11] Newell, Simon und Shaw

für Weiss geprüft und ermittelt für Stellung 7 den Wert von 875. Eine Überprüfung von Stellung 8 ist nicht mehr nötig, da Schwarz ohnehin die Stellung 3 präferieren wird. Der rechte Zweig wird analog abgearbeitet. Weiss ermittelt den Wert 87 für Stellung 10, der Wert für Stellung 13 braucht nicht mehr bestimmt zu werden, denn egal welchen Wert Stellung 13 auch hat, 87 ist der garantierte Wert für Schwarz in Stellung 9. Weiss in Stellung 1 wird also 800 vorziehen und damit Stellung 2 herbeiführen. Das einzige Problem das sich mit Alpha-Beta ergeben kann, ist die Reihenfolge in der die Züge untersucht werden. Im schlechtesten Fall, das heisst, wenn alle schlechten Züge zuerst geprüft werden, wird Alpha-Beta halt zu Mini-Max. In der Praxis bewährt sich Alpha-Beta jedoch erstaunlich gut und die Suchtiefe lässt sich damit verdoppeln.

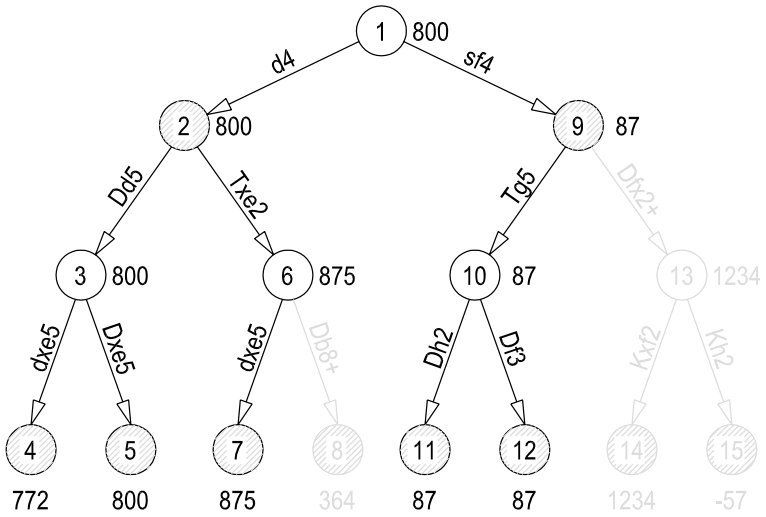

Abbildung 7: Spielbaum mit Alpha-Beta-Abschneidungen. Nur die relevanten Spielzüge werden weiterverfolgt.

Da sich die Suchtiefe mit Alpha-Beta verdoppelt, ist auch der Spielstärkezuwachs der Programme beträchtlich. Ein Programm mit Alpha-Beta, das 10 Halbzüge tief rechnet, weist nun bereits eine Spielstärke von rund 2'400 Elo Punkten auf und rückt damit bereits näher in die Region der Grossmeister vor. Über 2'400 Elo scheint sich die Kurve aber wieder etwas abzuflachen, das heisst pro Halbzug, der nun tiefer gerechnet wird, ergibt sich nur noch ein kleiner Spielstärkezuwachs. Ein Grund für dieses Phänomen dürfte das mangelnde positionelle Wissen der meisten Schachprogramme sein. Zwar ist die Mehrheit der Schachprogramme taktisch sehr stark und sie erkennen problemlos ein kompliziertes mehrzügiges Matt aber sehen zum Beispiel nicht, dass ein bestimmter Bauernzug, der kurzfristig einen kleinen Vorteil schafft, längerfristig zu einer Schwächung der Position und damit zum Verlust der Partie führen kann. Ein Computer, der Kasparovs Spielstärke aufweist, müsste nach Berechnungen von Thompson etwa 14 Halbzüge tief rechnen. Nachfolgend folgt ein Listing für einen Alpha-Beta-Algorithmus aus einem Schachprogramm. Die Verwandtschaft mit Mini-Max ist auf den ersten Blick ersichtlich. Auch diese Funktion nimmt die Brettstellung sowie drei Integerwerte als Argumente und liefert einen Integerwert zurück. Die Integerwerte alpha und beta geben Auskunft über den garantierten Rückgabewert eines Zuges, bzw. über den bestmöglichen Rückgabewert. Die Variable depth gibt Auskunft über die Tiefe, bis zu der die Suche ausgeführt werden soll. Der erste Aufruf der Alpha-Beta-Funktion erfolgt mit den Parametern -INFINITY und INFINITY für alpha und beta, wobei es sich um eine sehr kleine, bzw. eine sehr grosse Zahl handelt. Das anfänglich riesige Fenster reduziert sich zunehmend mit den

rekursiven Aufrufen der Funktion.

3.3.1 Alpha-Beta-Algorithmus in C-Notation:

(übernommen und stark modifiziert aus DOS [DM97])

```
int alpha_beta(struct position *p, int alpha, int beta, int depth) {
    int score, m, t, i;
    struct move list[MAXMOVES];

    if(depth == 0)                  /* Horizont erreicht */
        return evaluate(p);         /* Bewertungsfunktion wird aufgerufen */

    m = makemovelist(p, list);      /* m: anzahl legale Zuege */

    if(m == 0)
        checkfordraw(pos);          /* keine Zuege in dieser Pos. vorhanden */
                                    /* daher Remis weil Patt */
    score = -INFINITY;              /* sehr kleine Zahl */
    for(i = 0; i < m; i++) {        /* alle Zuege werden der Reihe nach */
                                    /* ausprobiert */
        domove(list[i], p);         /* Zug ausfuehren */
        t = -alphabeta(p, -beta, -alpha, depth-1);
                                    /* Rekursiver Aufruf */
                                    /* von AlphaBeta */
        undomove(p);                /* Zug wird zurueckgenommen */
    if(t > score)
        score = t;

    if(score >= beta)               /* Beta-Cutoff, Widerlegung des */
        break;                      /* letzten gegnerischen Zuges gefunden.*/
                                    /* Es brauchen keine weiteren  */
                                    /* Zuege untersucht zu werden */
    alpha = max(alpha, score);
        }

    return score                    /* Bewertung des besten Zuges */
}
```

Der Alpha-Beta Algorithmus hat sich insgesamt als recht erfolgreich erwiesen, haben doch wahrscheinlich noch immer die Mehrzahl der heutigen Schachprogramme Alpha-Beta, oder zumindest Varianten davon, implementiert. Aus verständlichen Gründen halten sich die Autoren der spielstärksten Programme natürlich etwas bedeckt, was die in ihren Programmen verwendeten Algorithmen betrifft.

3.3.2 Iterative Deepening

Die Alpha-Beta Funktion erhält als dritten Wert die Tiefe, bis zu der die Suche ausgeführt werden soll. Wie Tief soll nun die Suche aber überhaupt ausgeführt werden? Da sich je nach Komplexität der Stellung und der Anzahl Figuren auf dem Brett ja nur schlecht sagen lässt, wie lange die Suche dauern wird, wird meist einfach mal mit Tiefe 1 gesucht und nach Abschluss der Suche die Tiefe jeweils um eins erhöht. So wird auch sichergestellt, dass bei Zeitmangel überhaupt ein legaler Zug zur Verfügung steht, ehe die Suche zeitbedingt abgebrochen werden muss. Falls die Suche mit einem zu hohen Wert für die Tiefe startet, kann es passieren, dass gar nicht alle Züge in einer Stellung durchprobiert werden können und ein Matt (aus Zeitmangel) übersehen wird.

3.3.3 Ruhesuche/Quiescent Search

Unsere Alpha-Beta Funktion liefert uns nun einen Wert für eine Endstellung zurück, aber wir wissen nicht, ob beispielsweise beim Beenden der Suche unsere Dame irgendwo hängt (d.h. im nächsten Halbzug verloren geht). Aus diesem Grund versucht man, die resultierende Endstellung auch auf solche Gefahren hin zu untersuchen. Zugkombinationen bei denen Figuren abgetauscht werden oder aber eine Serie von drohenden Schachgeboten können ein Indiz für solche Bedrohungen sein. Falls nun für eine Endstellung solche Gefahren erkannt werden, dann werden diese Zugkombinationen untersucht, bis sich die Situation auf dem Brett weitgehend „beruhigt“ hat und erst dann wird die Suche beendet.

3.3.4 Nullmove

Im Schachspiel geht man generell davon aus, dass der Spieler, der am Zug ist, einen Vorteil hat. Um nun die Qualität einer Stellung zu überprüfen wird das Zugrecht umgekehrt und geprüft, welche Auswirkungen dies hätte. Wenn die Zugrechtsaufgabe keinen Vorteil für den Gegner ergibt, dann ist die eigene Stellung wahrscheinlich ziemlich gut. Anders gesagt, wenn der Gegner zwei Züge hintereinander ausführen könnte (was in Wirklichkeit natürlich unzulässig ist) und keinen wirklichen Vorteil daraus ziehen kann, muss seine Stellung ziemlich mies sein. Deshalb können wir die Suchtiefe verringern und sparen dadurch wertvolle Rechenzeit. In gewissen Stellungen ist Nullmove allerdings ziemlich gefährlich, da auch hier gewisse Suchbäume gar nicht erst untersucht werden, dafür aber Suchbäume untersucht werden, die so ja gar nie gespielt werden können, was katastrophale Auswirkungen auf den Spielverlauf haben kann. Im Endspiel verzichtet man aus diesen Gründen deshalb oftmals auf die Verwendung von Nullmove-Verfahren. Nachstehend folgt der Quelltext einer ziemlich „primitiven“ Implementation von Nullmove in Alpha-Beta. Man beachte, dass hier nicht getestet wird, in welcher Spielphase man sich gerade befindet.

```
#define R 3                          /* gebraeuchliche Werte fuer Nullmove */
                                     /* sind 2 oder 3 */

int alpha_beta(struct position *p, int alpha, int beta, int depth) {
    int score, m, t, i;
    struct move list[MAXMOVES];
    if(depth == 0
        return evaluate(p);


     /* Nullmove */
     if(imschach(p) == FALSE) {
         p->color = -(p->color);          /* Zugrecht wechselt */
         m = makemovelist(p, list);       /* Aufruf von Alpha-Beta mit red. Tiefe */
         for(i = 0; i < m; i++) {
             domove(list[i], p);
             t = -alphabeta(p, -beta, -alpha, depth-1-R);
             undomove(p);
         }
         p->color = -(p->color);
         if (t >= beta)                   /* falls t >= beta wird einfach Beta */
             return beta;                 /* zurueckgegeben, wir sparen eine */
      }                                   /* Menge Rechenzeit! */
     /* Ende Nullmove */

    m = makemovelist(p, list);
    if(m == 0)
        checkfordraw(pos);
```

```
    score = -INFINITY;
    for(i = 0; i < m; i++) {
        domove(list[i], p);
        t = -alphabeta(p, -beta, -alpha, depth-1);
        undomove(p);
    if(t > score)
        score = t;
    if(score >= beta)
        break;
    alpha = max(alpha, score);
        }
    return score
}
```

4 Bewertung

4.1 Material

Damit der Computer eine Vorstellung hat, welche Figuren sich untereinander abtauschen lassen oder mit Gewinn geschlagen werden können, muss der Wert der Figuren dem Computer mittgeteilt werden. Da der Computer Aussagen wie „ein Springer ist gleichviel Wert wie ein Läufer“ oder „eine Dame kann bedenkenlos gegen zwei Türme eingetauscht werden“ nicht versteht, müssen den Figuren Zahlenwerwerte zugewiesen werden, die über den Wert der Figur Auskunft geben. Folgende Werte haben sich dabei bewährt und finden in vielen Schachprogrammen Anwendung:

Figur	**Weiss**	**Schwarz**
Bauer:	1	-1
Springer:	3	-3
Läufer:	3.25	-3.25
Turm:	5	-5
Dame:	9	-9

Schwarze Figuren haben denselben Zahlenwert allerdings mit negativem Vorzeichen behaftet und da der König nicht geschlagen werden kann, erhält er auch keinen Wert zugewiesen. Die Materialwerte sind für die Bewertungsfunktion wichtig und können sich im Endspiel wieder verändern, da hier ein Bauer, der in eine Dame umgewandelt werden kann, natürlich mehr Wert hat als zwei Springer oder ein einzelner Läufer, mit denen der gegnerische König ja nicht Matt gesetzt werden kann.

Eine Bewertungsfunktion sucht nach dem virtuell ausgeführten Zug das Brett ab und addiert die Materialwerte zusammen. Positive Zahlen ergeben einen Materialvorteil für Weiss, während negative Zahlen einen Materialvorteil für Schwarz ergeben. Wer nach einer erfolgen Zugfolge mehr Material auf dem Brett hat, steht meistens klar besser da. Neben dem reinen Materialwert fliessen in die Bewertung natürlich zusätzlich noch weitere Kriterien, wie die Anzahl der beherrschten Felder, besetzte Felder im Zentrum, angegriffene Felder, verbundene Türme, Bauernstruktur und einiges mehr in die Bewertung ein. Trotzdem ist und bleibt aber der Materialwert häufig das entscheidende Kriterium bei der Zugwahl, denn häufig sind schon geringe Materialunterschiede ausschlaggebend für den Ausgang des Spiels und ein einzelner Bauer entscheidet oft über Sieg oder Niederlage. Je aufwendiger und komplexer die Bewertungsfunktionen der Schachprogramme sind, desto weniger Züge lassen sich in der gegebenen Zeit allerdings auch überprüfen. Es gilt hier also einen goldenen Mittelweg zwischen „Schachwissen “und Geschwindigkeit zu finden.

Nachstehend der Quellcode einer rein materialistisch agierenden Bewertungsfunktion. Es werden einfach sämtliche Figurenwerte addiert, wobei positive Zahlen einen Materi-

alvorteil für Weiss bedeuten,während negative Zahlen einen Materialvorteil für Schwarz bedeuten.

```
int eval_material (struct position *p)
{

  int i, materialw;

  for (i = 21, materialw = 0; i < 99; i++)
    {
      if (p->b1[i] == Bauer_weiss)
        materialw = materialw + 100;
      if (p->b1[i] == Bauer_schwarz)
        materialw = materialw - 100;
      if (p->b1[i] == Springer_weiss)
        materialw = materialw + 300;
      if (p->b1[i] == Springer_schwarz)
        materialw = materialw - 300;
      if (p->b1[i] == Laeufer_weiss)
        materialw = materialw + 325;
      if (p->b1[i] == Laeufer_schwarz)
        materialw = materialw - 325;
      if (p->b1[i] == Turm_weiss)
        materialw = materialw + 500;
      if (p->b1[i] == Turm_schwarz)
        materialw = materialw - 500;
      if (p->b1[i] == Dame_weiss)
        materialw = materialw + 900;
      if (p->b1[i] == Dame_schwarz)
        materialw = materialw - 900;
    }

  return materialw;
}
```

4.2 Tabellen

Beim Schachspiel geht es häufig nicht nur darum, kurzfristige Materialgewinne zu erzielen, sondern auch eine stabile Position auf dem Brett zu erreichen. Damit bekunden die meisten Schachprogramme aber ziemliche Probleme. Die erreichten Suchtiefen reichen dazu ganz einfach noch nicht aus, um kleine positionelle Feinheiten ausrechnen zu können und so machen halt einige Programme ausgiebig Jagd auf feindliche Bauern, vernachlässigen dabei die Entwicklung der eigenen Figuren und schwächen dadurch natürlich die eigene Position. Gegen starke menschliche Spieler oder Programme mit etwas positionellem Grundwissen, geht das natürlich meist nicht lange gut. Die meisten Programme erhalten deshalb Unterstützung durch Tabellen, die darüber Auskunft geben, auf welche Felder die Figuren mit Vorteil zu platzieren sind. Zum Beispiel bewertet die Bewertungsfunktion eigene Springer oder Läufer, die im Zentrum stehen, mit einem höheren Wert, als wenn diese wirkungslos auf der eigenen Grundlinie oder irgendwo am Rand stehen. Die Bewertungsfunktion konsultiert zu diesem Zweck besagte Tabellen und das Programm versucht eigene Figuren nach Möglichkeit auf diese Felder zu stellen und feindliche Figuren möglichst von diesen Feldern fernzuhalten.

Nachstehend der Quellcode einer Springer Tafel (8x8) für Weiss. Da die Springer im Zentrum des Brettes ihre Stärke am besten entfalten, erhalten Springer auf Feldern im

Zentrum eine höhere Bewertung, während Springer, die irgendwo am Rand stehen oder noch gar nicht entwickelt sind natürlich schlechter bewertet werden.

```
int Springer_weiss[64] = {
        -10, -10, -10, -10, -10, -10, -10, -10,
        -10,   0,   0,   0,   0,   0,   0, -10,
        -10,   0,   5,   5,   5,   5,   0, -10,
        -10,   0,   5,  10,  10,   5,   0, -10,
        -10,   0,   5,  10,  10,   5,   0, -10,
        -10,   0,   5,   5,   5,   5,   0, -10,
        -10,   0,   0,   0,   0,   0,   0, -10,
        -10, -30, -10, -10, -10, -10, -30, -10
};
```

Die Verwendung solcher Tabellen hat sich als probabtes Mittel erwiesen, die Figuren richtig zu entwickeln. In der Endphase eines Spiels werden natürlich meist andere Felder wichtiger für den Spielverlauf und die Tabellen werden in dieser Spielphase dann auch in vielen Programmen gar nicht mehr verwendet.

5 Einige Begriffe

Einige der anschliessend aufgeführten Begriffe werden zwar in dieser Maturaarbeit gar nicht erwähnt, tauchen aber im Zusammenhang mit Computerschach so häufig auf, dass sie hier trotzdem kurz erklärt werden.

Forward Pruning

Auch hier geht es darum, den Verzweigungsfaktor der Suchbäume zu reduzieren. Es werden nur die wirklich „erfolgsversprechenden“ Zweige weiterverfolgt. Die Suche im Spielbaum wird dadurch tiefer, dafür verringert sich natürlich die Breite der Suche. Im Gegensatz zum Alpha-Beta-Verfahren ist Forward Pruning aber kein exaktes Verfahren mehr, das heisst, es können sich hier durchaus auch Fehler einschleichen, da die verwendeten Verfahren meist nur Daumenregeln sind, die zwar oftmals zum Erfolg aber halt auch ab und zu ins Verderben führen können.

Hashtables/Transposition Tables

Die gleiche Brettstellung kann durch verschiedene Zugfolgen erreicht werden. Um nun keine wertvolle Rechenzeit zu verschwenden, werden in den meisten Schachprogrammen sogenannte Transposition Tables (auch als Hashtables bezeichnet) verwendet. Kommt eine Stellung während eines Spieles auf das Brett, die bereits zuvor einmal berechnet wurde, so kann man sich das erneute Berechnen der aussichtsreichsten Folgezüge ersparen, wenn die bereits berechneten und abgespeicherten Zugfolgen aus dem Speicher abgerufen werden können. Dadurch kann Rechenzeit eingespart werden.

Pondering/Permanent Brain

Falls zwei Computer gegeneinander spielen, kann die Zeit in der der gegnerische Computer seinen Antwortzug berechnet, dazu genutzt werden, seinen wahrscheinlichsten Antwortzug zu berechnen und dann einen darauf entsprechenden Gegenzug zu berechnen. Falls der erwartete Zug gespielt wird, haben wir eine Menge Zeit gewonnen, die entweder dazu genutzt werden kann, diese Variante tiefer zu untersuchen, oder aber den Gegner unter Zeitdruck zu setzten, indem der berechnete Zug direkt ausgespielt wird.

Engines

Damit wird das Programm bezeichnet, das die Züge erzeugt, bewertet und auswählt. Die engine ist, um die Bedienung benutzerfreundlich zu gestalten, meist in eine graphische Oberfläche eingebunden. Wenn in dieser Maturaarbeit von einem Schachprogramm die Rede ist, dann ist damit die Engine gemeint.

Eröffnungsbücher

Die Eröffnung, das heisst die ersten Züge in einem Schachspiel, bestimmen oftmals den Charakter einer ganzen Partie. In den hunderten von Jahren, seit Schach gespielt wird, haben sich deshalb viele Spieltheoretiker mit den aussichtsreichsten Eröffnungen befasst. Um nun diese Erkenntnisse auch für den Computer nutzbar zu machen und zu verhindern, dass die Rechner diese Anfangszüge immer wieder neu berechnen müssen, verfügen die meisten Schachprogramme über Eröffnungsbücher, die es ermöglichen automatisch auf einen bestimmten Zug mit einem entsprechenden Gegenzug zu antworten, ohne dabei auch nur eine Sekunde Rechenzeit in Anspruch zu nehmen. Da in diesen Eröffnungsdatenbanken oftmals Millionen von Zügen gespeichert sind, können die Schachprogramme oftmals eine Vielzahl von Zügen ausführen, ehe sie überhaupt zu Rechnen beginnen.

Tablebase/Endspieldatenbank

Da beim Endspiel im Schach oft nur noch ein paar wenige Figuren auf dem Brett stehen, reduziert sich die Anzahl der möglichen Züge natürlich stark. Endspiele mit maximal 6 Steinen auf dem Brett lassen sich aber exakt lösen. Bei Bedarf werden dann im Endspiel, wenn maximal noch 5 bis 6 Figuren auf dem Brett stehen (oder aber die Möglichkeit besteht, die Anzahl der Figuren auf 5 oder 6 Stück zu reduzieren), einfach die entsprechenden Datenbanken konsultiert und das Schachprogramm kann das Endspiel anschliessend „perfekt“ abwickeln, remis anbieten oder aufgeben, wenn die Niederlage unvermeidbar ist.

6 Zusammenfassung

War die Schachprogrammierung in ihren Anfängen ein Betätigungsfeld für Forscher, die sich daraus Erkenntnisse über menschliche Intelligenz erhofften, ist das Ganze heute eher zu einem Geschäft geworden. Zudem befassen sich viele Hobbyprogrammierer mit der Schachprogrammierung, wie die dutzenden von Schachprogrammen beweisen, die zumeist kostenlos im Internet erhältlich sind. Das ursprüngliche Ziel, aus der Schachprogrammierung Erkenntnisse über künstliche Intelligenz zu erlangen, hat sich allerdings nicht erfüllt, da sich die relativ simpel gestrickten Schachprogramme allen anderen Ansätzen weitaus überlegen zeigten.

Obwohl die meisten Schachprogramme bereits viel zu stark für die Mehrzahl der Schachspieler sind, kommen immer wieder neue noch leistungsfähigere Programme auf den Markt. Mangels geeigneter Gegner lässt man dann halt die Rechner gegeneinander spielen oder reduziert die Spielstärke der Programme, um auch mal zu gewinnen. Erst durch diese Option, die Spielstärke zu reduzieren, werden die meisten Schachprogramme überhaupt sinnvoll für Anfänger und schwächere Spieler. Als Lerninstrument für Anfänger sind aber die oft mit aufwendiger graphischer Oberfläche ausgestatteten Programme meist ganz gut geeignet. Vor allem lassen sich damit z.B. elementare Endspiele trainieren oder aber eigene Partien auf taktische Fehler hin untersuchen. Das Spiel Mensch gegen Mensch wird aber wohl kaum etwas von seinem Reiz verlieren, auch wenn der Schachweltmeister in ferner (oder auch nicht so ferner) Zukunft wohl ein Computer sein wird. Schliesslich gibt es auch noch immer Pferderennen, obwohl doch mittlerweile jedes Auto einem Rennpferd davonfährt. Viele Menschen ziehen das Spiel gegen Menschen, dem Spiel gegen den Computer ohnehin vor, wie auch einige Schachserver beweisen, wo täglich tausende Menschen übers Internet Schach spielen.

Literatur

[Bot82] M. M. Botwinnik. *Meine neuen Ideen zur Schachprogrammierung.* Berlin: Springer, 1. edition, 1982.

[CS] William G. Chase and Herbert A. Simon. Perception in chess. 4.

[DM97] Dr. Christian Donninger and Dr. Klaus Manhart. Das Spiel der Könige. *DOS*, 4, 1997.

[Hya04] Dr. Robert Hyatt. Crafty, ein leistungsfähiges, bitboardbasiertes Schachprogramm. *http://www.cis.uab.edu/hyatt/index.html*, 2004.

[Sha50] Claude E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41, 1950.

[vR] Erik van Reem. Der Traum vom Computerschach, Geschichte des Computerschachs. *http://www.rochadekuppenheim.de/coko/computerschach.htm.*

[Zib] Frank Zibi. An introduction to bitboards by the programmer of ZChess. *http://www.computerschach.de/freeware/bitboards.htm.*