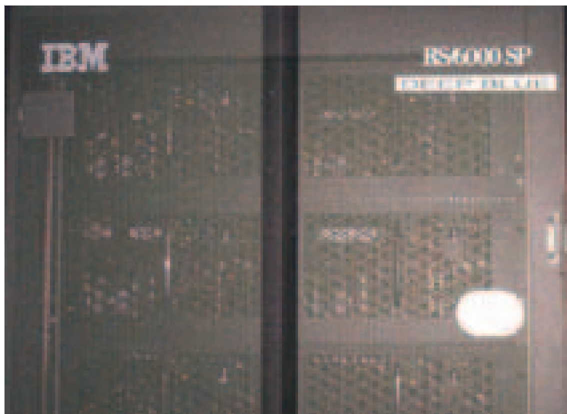
Johannes Zwanzger

Das Geheimnis der Schachprogramme



Am 11. Mai 1997 ereignete sich in New York Historisches: mit Garry Kasparovs 2,5 – 3,5 Niederlage gegen den IBM-Großrechner „Deep Blue“ verlor zum ersten Mal in der Geschichte der amtierende Weltmeister in einem Wettkampf unter regulären Turnierbedingungen gegen einen Schachcomputer. Obwohl Kasparov in diesem Match offensichtlich deutlich unter Normalform agierte und sich in der 6. Partie, für ihn völlig untypisch, sogar einen katastrophalen Eröffnungsfehler geleistet hatte, der den Verlust dieser entscheidenden Partie alsbald nach sich zog, war für viele klar, dass nun die Zeit gekommen war, in der die Computer auch bald noch die letzte „menschliche Bastion“ im Sturmlauf erobern würden: die sog. „künstliche Intelligenz“ schicke sich an, der Vorherrschaft des menschlichen Geistes ein für allemal ein Ende zu setzen... Inzwischen sind fast sieben Jahre vergangen und die erste Aufregung vorbei; „Deep Blue“ wurde in seine Einzelteile zerlegt, Kasparov ein Revanchematch verwehrt. Dafür haben in letzter Zeit mehrere Wettkämpfe von Weltklasseschachspielern gegen führende Schachprogramme (trotz des Hardwarenachteils mutmaßlich stärker als „Deep Blue“) gezeigt, dass die Menschheit, was Schach betrifft, noch lange nicht abgehängt ist: die letzten acht endeten allesamt unentschieden.[1] Dieser Artikel soll einen groben Überblick über die Entstehungsgeschichte, Funktionsweise und den Leistungsstand heutiger Schachprogramme geben.

Abbildung oben: Foto des IBM-Großrechners, der 1997 Kasparov besiegte.

1 Die Vorläufer

Man sollte annehmen, dass es Schachprogramme frühestens seit der Entwicklung des ersten Computers geben könne. Umso überraschter wird man feststellen, dass bereits im 18. Jahrhundert ein Schachautomat für Aufsehen sorgte, der sog. „Türke“ von Hofrat Wolfgang von Kempelen; er verdankt seinen Namen einer an einem Schachtisch angebrachten Figur, welche in türkische Gewänder gehüllt war. In den Tisch war allerhand „Technik“ in Form von Zahnrädern und ähnlichem eingebaut. Diese Maschine war scheinbar in der Lage, selbstständig Züge zu ersinnen und auszuführen; eine kurze Gewinnpartie des „Türken“ gegen Napoleon ist sogar überliefert.[2] Freilich steckte, wie Kempelen selbst zugab, lediglich ein Trick dahinter; offensichtlich musste sich ein kleinwüchsiger Schachspieler in dem Tisch versteckt halten, dem die Stellung auf dem Brett durch mit den Figuren magnetisch gekoppelte Steine ins Innere übertragen wurde und der über geschickt konstruierte Mechanik seine Züge ausführen konnte. Obwohl der Tisch Mitte des 19. Jahrhunderts einem Brand zum Opfer fiel, kann man seit März 2004 eine Rekonstruktion im Heinz-Nixdorf-Museum Paderborn bewundern.[3] Übrigens: Das deutsche Adjektiv „getürkt“ hat seine Ursprünge

[1]) Das letzte derartige Match war 2003 der Wettkampf Fritz - Kasparov, Endstand 2-2
[2]) unter www.chesslive.de findet man bei einer Suche nach “Napoleon“ die Partie: Napoleon-The Automaton, Wien 1809
[3]) www.wdr.de/themen/computer/1/schachtuerke

in genau dieser Automatenattrappe. Eine nette Geschichte, die jedoch streng genommen mit Schachprogrammierung nicht das Geringste zu tun hat. Trotzdem gab es in der Vorcomputerära in diesem Zusammenhang zumindest zwei erwähnenswerte Persönlichkeiten: Zum einen den spanischen Ingenieur Torres y Quevedo, der 1890 einen elektromechanischen Roboter konstruierte, welcher das Mattsetzen mit Turm und König gegen König bewerkstelligen konnte – eine Aufgabe, die zumindest ungeübten Spielern durchaus Schwierigkeiten bereiten kann.

Das erste „vollwertige“ Schachprogramm schrieb dann während des zweiten Weltkriegs Alan Turing, der damals übrigens an der Entschlüsselung des Enigma-Codes beteiligt war. Ausgeführt wurden die Operationen mangels geeigneter Hardware nicht auf einem Computer – sondern von Turing selbst per Hand! Umso erstaunlicher, dass er auf diese Weise 1952 tatsächlich unter enormem Zeitaufwand eine Partie gegen den Hobbyspieler Alick Glennie zu Ende bringen konnte - wenngleich sein Programm verlor.[4]

Museumsfoto des Roboters von Quevedo

Zeichnung des Schachautomaten von Kempelen

2 Grundlagen eines Schachprogramms

Auch als die ersten Computer bereits entwickelt waren, wurde die Diskussion über die günstigste Arbeitsweise eines Schachprogramms zunächst nur theoretisch geführt. Die grundlegendsten Erkenntnisse stammen vor allem von Claude Shannon, einem Mathematikprofessor aus Amerika. Es kristallisierte sich heraus, dass ein Schachprogramm notwendigerweise folgende Komponenten enthalten sollte: Zunächst einmal muss das Programm natürlich die Schachregeln beherrschen, das heißt, in einer beliebigen Stellung alle regelkonformen Züge ermitteln können. Die hierfür zuständige Routine wird als **Zuggenerator** bezeichnet und ist nach Erfahrung des Autors der mit Abstand fehleranfälligste (und zugegebenermaßen langweiligste) Teil bei der Programmierung. Dennoch spielt die Implementierung des Zuggenerators eine große Rolle, da die Ausführungsgeschwindigkeit

[4]) Diese Partie ist, zusammen mit einigen interessanten Aspekten zu Turings Programm, unter http://www.schachcomputer.at/gesch3.htm vermerkt

Das Geheimnis der Schachprogramme

des Programms nicht unerheblich von deren Effizienz abhängt. Als nächstes muss dem Programm gesagt werden, woran es erkennen kann, welche Positionen günstig und welche ungünstig sind; dies erledigt die sogenannte **Bewertungsfunktion**, welche jeder möglichen Stellung eine ganze Zahl **W** zuordnet, die für gewöhnlich den Wert der Position aus weißer Sicht repräsentiert. Einer der Hauptfaktoren für die Bewertung einer Stellung ist naturgemäß das Materialverhältnis am Brett: wer mehr oder wertvollere Figuren besitzt, wird in aller Regel im Vorteil sein. Es ist üblich geworden, die Werte in (hundertstel) sog. „Bauerneinheiten" zu messen. Wenn also Schwarz in einer Stellung bei sonst ausgeglichener Lage einen Bauern mehr auf dem Brett hat, wäre **W** = -100. Aber natürlich gibt es noch andere wichtige Bewertungskriterien: die Sicherheit der beiden Könige etwa, die Aktivität der Figuren, die Bauernstruktur etc. können eine so große Rolle spielen, dass ein materieller Nachteil kompensiert wird; in der Regel wird für jeden dieser Teilbereiche ein Wert bestimmt und am Ende einfach alles zur Gesamtbewertung aufsummiert. Insgesamt besteht also die Kunst bei der Implementierung der Bewertungsfunktion darin, erstens möglichst viele relevante Stellungsmerkmale zu berücksichtigen und zweitens so zu gewichten, dass das Endresultat die „schachliche Realität" möglichst gut wiedergibt. Zu beachten ist, dass es hier lediglich um eine *statische* Bewertungsfunktion geht, d.h. sie erfolgt ohne Rücksichtnahme auf irgendwelche Zugfolgen. Damit ist klar, dass die auf diese Weise gewonnenen Schätzwerte sehr grob und manchmal gar völlig falsch sind; man denke nur an eine materiell ausgeglichene Stellung, in welcher der Weiße gerade die Damen abtauscht: nach dem Schlagen der schwarzen Dame hat er erst einmal großen Materialvorteil, die statische Bewertungsfunktion würde also einen sehr hohen (aus schwarzer Sicht schlechten) Wert zurückliefern. Dass die weiße Dame sofort im nächsten Zug wiedergenommen werden kann, bleibt unberücksichtigt. Natürlich kann man versuchen, auch noch angegriffene und gedeckte Figuren bei der Bewertung zu berücksichtigen, um Fälle wie den gerade geschilderten abzufangen. Aber spätestens bei noch komplizierteren Zugfolgen wird man nicht umhin kommen, dass das Programm auch einmal etwas berechnen muss. Hierfür wird ein **Suchbaum** durchlaufen. Er soll hier zunächst in einer vereinfachten Variante dargestellt werden, eine entscheidende Verbesserung wird im nächsten Abschnitt beschrieben. Das Verfahren ist als sog. „Tiefensuche" in der Informatik bekannt: ausgehend von der gerade am Brett befindlichen Stellung ermittelt das Programm mittels des Zuggenerators alle regelkonformen Züge und führt „im Geiste" den ersten möglichen davon aus; in der neu entstandenen Stellung werden nun wiederum alle Züge bestimmt, der erste ausgeführt usw., bis zu einer bestimmten Tiefe **t**. Ist diese erreicht, schätzt man die entstandene Stellung mittels der Bewertungsfunktion ab; anschließend wird der letzte Zug zurückgenommen und nacheinander alle Alternativmöglichkeiten durchprobiert, wobei hier noch jeweils unmittelbar im Anschluss die Bewertungsfunktion aufgerufen wird. Sind alle Möglichkeiten untersucht, wird auch der vorletzte Zug zurückgenommen und nun dessen Alternativen getestet, worauf genau das gleiche Verfahren einsetzt (d.h. Ausprobieren aller möglichen Züge und Aufruf der Bewertungsfunktion). Macht man so weiter, hat man am Ende alle Stellungen besichtigt, die sich nach t Halbzügen[5] ergeben könnten. Wurde für jede dieser Stellungen die Bewertung und der „Pfad" (d.h. die Züge, die zur Stellung führten) gespeichert, kann man sich einen Baum (wie in der Abbildung unten) generieren:
Die Knoten stellen die während des Durchlaufens auftretenden Stellungen dar, die Kanten die Züge, durch welche jeweils die Stellung am oberen Ende der Kante in die am unteren Ende übergeht. Der Baum besteht offensichtlich aus vier horizontalen **Stufen**, der einzelne Knoten auf der obersten Stufe entspricht der Ausgangsstellung, diejenigen auf der untersten Stufe den Stellungen, bei denen die Bewertungsfunktion aufgerufen wird (im Folgenden als **Endknoten** bezeichnet). Geht man davon aus, dass Weiß zu Beginn am Zug ist, so

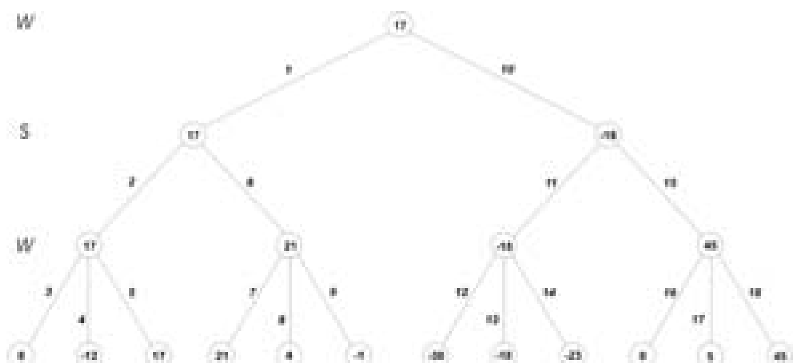


Fig. 1: Beispiel für einen Suchbaum, die kursiv gedruckten Zahlen an den Kanten geben die Durchlaufreihenfolge des Algorithmus an

[5]) *Ein „Zug" dagegen besteht streng genommen aus je einem „Halbzug" von Schwarz und Weiß; oft ist hiermit aber auch das Gleiche wie bei „Halbzug" gemeint*

erfolgt der zweite Zug von Schwarz, der dritte wiederum von Weiß. Auf der vierten Stufe wird nicht mehr gezogen, sondern lediglich die Stellung bewertet. In allen untersuchten Zugfolgen macht also Weiß den letzten Zug, bevor die Bewertungsfunktion aufgerufen wird. Das Ziel ist jetzt, herauszufinden, welche der beiden Möglichkeiten er im ersten Zug wählen sollte, um die (bei optimalem eigenem und gegnerischem Spiel) bestmögliche Stellung für sich zu erreichen. Dies gelingt, indem man den Baum von hinten „aufrollt“: Zu Beginn kennt man ja nur die Schätzwerte für die Endknoten; da aber Weiß mit seinem letzten Zug stets den größtmöglichen Wert anstreben wird, kann man nun auch allen Knoten auf der vorletzten Stufe einen Wert zuordnen, nämlich das Maximum aus allen Knoten der untersten Stufe, die mit diesem Knoten durch eine Kante verbunden sind. Diese Stellungen entstehen alle nach einem vorhergehenden Zug des Schwarzen; der wird natürlich so spielen, dass eine Stellung mit möglichst niedrigem Wert entsteht. Folglich lassen sich nun auch die Knoten auf der zweiten Stufe bewerten, indem man einfach das Minimum der Werte aus den unmittelbar folgenden Knoten (die ja im Schritt vorher bestimmt wurden) bildet. Analog erfolgt der letzte Schritt: der Weiße wird nun aus seinen beiden Möglichkeiten in der Anfangsstellung die auswählen, welche zur Stellung mit dem höchsten Wert führt; im Beispiel würde er sich also für den linken Teilbaum entscheiden. Wohlgemerkt: der rechte Ast wird vermieden, obwohl sich in ihm der Endknoten mit dem höchsten Wert aus weißer Sicht (nämlich 45) befindet; das ist aber auch vernünftig, denn um in diesen Knoten zu gelangen, müsste Schwarz im Zug vorher die schlechtere Variante in Gestalt des rechten Teilastes wählen. Da wir aber von bestmöglichem Spiel beider Seiten ausgehen, gibt es keinen Grund anzunehmen, dass er dies tun würde. Spielt er stattdessen den Zug zum linken Teilast, käme Weiß mit einem Wert von -18 sogar in Nachteil. Natürlich lässt sich das geschilderte Verfahren auf Bäume jeder Höhe und beliebigen Verzweigungsgrades ausweiten und ist in keiner Weise nur auf Schach beschränkt. Prinzipiell ist damit jedes sog. **„Zweispieler-Nullsummenspiel“** (ein Spiel, bei dem zwei Spieler gegeneinander spielen und jeder Vorteil des einen ein entsprechender Nachteil für den anderen ist; daher die Bezeichnung „Nullsumme“) behandelbar. Es muss lediglich der Zuggenerator und die Bewertungsfunktion an das jeweilige Spiel angepasst werden. Dieser Algorithmus erhielt wegen des abwechselnden Maximierens und Minimierens der Werte durch Weiß bzw. Schwarz den schönen Namen **„Minimax“**.

Bei der Implementierung ist es freilich nicht nötig, den Baum als ganzes im Speicher zu haben. Vielmehr kann das Maxi- bzw. Minimieren bereits beim Durchlaufen geschehen; dann müssen zu jedem Zeitpunkt lediglich die gerade untersuchte Stellung, der zu ihr führende Pfad (samt evtl. nötiger Information zum Zurücknehmen der Züge) und die Zuglisten zu den auf dem Pfad vorausgehenden Knoten gespeichert werden. Zur leichteren Formulierung ist es günstig, immer nur zu maximieren und dafür bei Werten, die von Knoten höherer Stufe zurückkommen, das Vorzeichen zu negieren. Mithilfe einer Rekursion lässt sich „Minimax“ im C-Pseudocode dann überraschend kurz formulieren (Kasten rechts oben).

```
//außerhalb definierte Variablen:
//depth: Rechentiefe
//checkMateValue: Konstante für Mattwert
//whiteToMove: zeigt Zugrecht an

int minimax(int c){
    int bestValue;
    if (c>=depth){ //bewerten
        if (whiteToMove)
            bestValue=evaluatePosition();
        else //dann Vorzeichen drehen
            bestValue=-evaluatePosition();
    }
    else{
        //falls kein Zug möglich ->Mattwert
        bestValue=-checkMateValue+c;
        for („each possible move m“){
            executeMove(m);
            int temp=-minimax(c+1);
            if (temp>bestValue)
                bestValue=temp;
            takeBackMove(m);
        }
    }
    return bestValue;
}
```

Aufgerufen würde der Algorithmus dann einfach mit „minimax(0)“. Selbstverständlich muss man, um wirklich spielen zu können, noch irgendwie dafür sorgen, dass für die Ausgangsstellung nicht nur der Wert, sondern auch der dazugehörige Zug zurückkommt. Wie man an der Bezeichnung *checkMateValue* erkennen kann, ist hier auch schon eine kleine schachspezifische Eigenheit berücksichtigt worden: das Matt, welches vorliegt, wenn der König der am Zug befindlichen Partei bedroht ist und sich dies mit dem nächsten Zug nicht mehr abwenden lässt. Erkannt würde es bei dieser Implementierung dadurch, dass die mattgesetzte Partei über keinerlei legale Züge mehr verfügt, so dass der Mattwert (der schlechtest mögliche überhaupt) zurückgeliefert wird. Durch die Addition der Variable c, welche einfach die seit der Ausgangstellung erfolgten Züge zählt, wird zusätzlich die Information mitgegeben, wie lang die Zugfolge zum Matt ist. Könnte der Rechner also in einer bestimmten Stellung ein Matt auf mehrere Weisen erreichen, würde er den kürzesten Weg wählen. Das Patt, bei dem die Partei am Zug auch keine legalen Züge mehr hat, ihr König aber nicht angegriffen ist, muss man noch extra behandeln, doch das soll hier keine Rolle spielen.

Das Geheimnis der Schachprogramme

3 Alpha-Beta-Suche

Theoretisch, entsprechende Geduld und Lebensdauer vorausgesetzt, wäre man mit Hilfe des Minimax-Algorithmus bereits fähig, jedes Zweispieler-Nullsummenspiel, in dem es nur endlich viele mögliche Zugfolgen gibt, auszuanalysieren, also u.a. auch Schach (denn die sog. „Fünfzig-Züge-Regel" besagt, dass nach einer Serie von fünfzig Zügen beider Seiten, in denen nichts geschlagen und kein Bauer gezogen wurde, die Partie mit remis endet). Praktisch ist dies bei einem derart komplexen Spiel vollkommen unmöglich: man gehe von einer gewöhnlichen Schachpartie aus, die in der Regel etwa 80 Halbzüge dauert; im Mittel besitzt jede Partei pro Stellung vielleicht 40 Zugmöglichkeiten. Minimax müsste dann einen Baum mit 40^{80} Endknoten durchlaufen, was selbst „Deep Blue" mit einer Rechenleistung von etwa 200 Millionen Knoten pro Sekunde mehr als 10^{112} Jahre beschäftigen würde. Bereits für eine Rechentiefe von fünf Halbzügen bräuchte ein heutiges Spitzenprogramm, das auf derzeit handelsüblichen Rechnern vielleicht eine Million Stellungen pro Sekunde berechnen kann, ca. 100 Sekunden. De facto kommt es in dieser Zeit aber auf Rechentiefen von etwa 14-15 Halbzügen; das verdankt es zu einem großen Teil dem folgenden „Trick": Machen wir uns noch einmal klar, was wir eigentlich wollen. Aus einer vorgegebenen Stellung heraus soll der Zug ermittelt werden, der nach einer Minimax-Suche mit einer bestimmten Tiefe die gemäß der Bewertungsfunktion beste Position verspricht. Das ist etwas anderes als jeden möglichen Zug in einer Stellung genau bewerten zu wollen; es interessiert ja nur der beste. Was beim ersten Lesen vielleicht lediglich spitzfindig klingt, birgt gewaltiges Einsparungspotential in sich: Nehmen wir an, das Programm hat in einem beliebigen Knoten K_0 des Baumes bereits einige Züge untersucht und für den bisher besten den Wert w_0 aus Sicht der am Zug befindlichen Partei **A** ermittelt. Gemäß des Minimax-Schemas untersucht es nun die nächste Alternative; sollte dann bei der Suche in diesem Teilbaum an irgendeiner Stelle der Gegner **B** die Möglichkeit haben, einen Zug **z** mit einer Bewertung w_z, die aus Sicht von **A** nicht besser als w_0 ist, zu spielen, so weiß man, dass einer der vorangegangenen Züge von **A** (ab dem Knoten K_0) bereits nicht optimal war, denn den Wert w_0 kann er ja dadurch erreichen, dass er in K_0 den zu w_0 gehörigen Zug wählt. Es ist dann an dieser Stelle unnötig, noch weitere Gegenzüge von **B** auszuprobieren; das vorhergehende Spiel von **A** kann bereits durch **z** widerlegt werden, ob es noch stärkere Züge gibt, ist uninteressant. Folglich würde an dieser Stelle der Wert w_z als **obere Schranke** für den Wert des letzten Zuges von **A** zurückgegeben und die nächste Alternative zu diesem getestet. Der Wert w_0 kann also bei allen Folgeknoten von K_0 als sicherer Wert für die Partei **A** angesehen werden. Bei den meisten Knoten, die tiefer im Baum liegen, wird es so sein, dass solche sicheren Werte sowohl für Weiß als auch für Schwarz existieren, da für beide bereits in übergeordneten Stellungen alternative Möglichkeiten vollständig berechnet wurden; bei der Untersuchung eines Knotens bezeichnet man den zu diesem Zeitpunkt sicheren Wert für die Partei am Zug als **alpha-**, den für den Gegner als **beta-**Wert, wobei alpha stets kleiner beta ist. Dementsprechend heißt dieses Verfahren **alpha-beta-Suche**, die Widerlegung eines Zuges durch einen Gegenzug und die damit verbundene Abschneidung des Suchbaumes „Cutoff". Der zu untersuchende Baum aus Fig.1 verkleinert sich damit wie in der Abbildung unten zu sehen.

Es sei an dieser Stelle noch einmal betont, dass die alpha-beta-Suche erlaubt, **risikolos** bestimmte Äste des Baumes wegzulassen, d.h. es wird genau der gleiche Zug mit dem gleichen Wert als bester ermittelt wie bei der vollständigen Minimax-Suche. Die Einsparung dagegen ist ganz enorm und hängt entscheidend von der Zugsortierung ab: man kann sich leicht überlegen, dass es ungünstig ist, wenn schlechte Züge als erste untersucht werden, denn diese liefern nur niedrige sichere Werte bzw. bewirken nur selten einen Cutoff. Der Idealfall wäre, wenn man in jeder Stellung stets mit dem besten Zug beginnt. Natürlich kann man diesen vorher nicht wissen (dafür führt man den Algorithmus ja erst durch), aber zumindest „ahnen": im Schach beispielsweise ist die Wahrscheinlichkeit sehr hoch, dass sich ein Schlagzug als bester herausstellt. Folglich wird man, ceteris paribus,

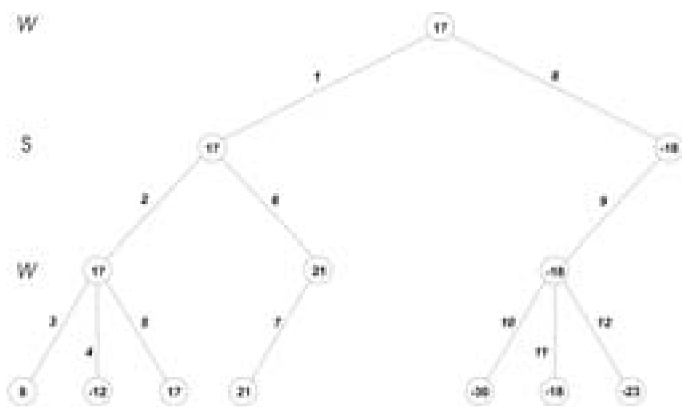


Fig.2: Reduktion des Baumes durch die alpha-beta-Suche. Widerlegung von Zug (6) durch (7), denn Zug (2) liefert 17 zurück (aus schwarzer Sicht besser als 21); Widerlegung von Zug (8) durch Zug (9), denn Zug (1) sichert bereits einen Wert von 17.

Schlagzüge als erstes ausprobieren, wobei man mit demjenigen beginnt, der die höchstwertige gegnerische Figur beseitigt. Es gibt noch viele weitere Faustregeln zur Zugsortierung im Schach, doch das würde den Rahmen dieses Artikels sprengen; für andere Spiele lassen sich im übrigen in der Regel auch solche Kriterien finden. Nur um ein Gefühl für die Einsparung durch die alpha-beta-Suche zu vermitteln, hier kurz (ohne Begründung) die Zahl zu bewertender Endknoten n beim oben skizzierten Idealfall für einen Baum der Höhe d und (als konstant angenommenem) Verzweigungsfaktor z:

für d **ungerade**:

$$n = z^{\frac{d+1}{2}} + z^{\frac{d-1}{2}} - 1$$

für d **gerade**:

$$n = 2 \cdot z^{\frac{d}{2}} - 1$$

Wie man den Formeln entnimmt, steigt die Zahl zu bewertender Endknoten bei Erhöhung der Suchtiefe um einen Halbzug im Mittel um den Faktor $\sqrt{z}$; zur Erinnerung: mit dem einfachen Minimax-Algorithmus ist dieser gleich z! Grob gesprochen kann man also mit der Alpha-Beta-Suche bei gleicher Endknotenzahl etwa doppelt so tief rechnen wie mit Minimax. Im Pseudocode dagegen liest sich Alpha-Beta nicht viel komplizierter, wie der Kasten rechts oben zeigt.
Der Aufruf von der Ausgangsstellung erfolgt durch

„`alphaBeta(0,-checkMateValue,+checkMateValue)`“.

```
//außerhalb definierte Variablen:
//depth: Rechentiefe
//checkMateValue: Konstante für Mattwert
//whiteToMove: zeigt Zugrecht an

int alphaBeta(int c,int a,int b){
    if (c>=depth){ //bewerten
        if (whiteToMove)
            return evaluatePosition();
        else //dann Vorzeichen drehen
            return -evaluatePosition();
    }
    else{
        //Mattwert ist ggf. sicher
        a=max(a,-checkMateValue+c);
        for („each possible move m“){
            executeMove(m);
            int temp=-alphaBeta(c+1,-b,-a);
            takeBackMove(m);
            if (temp>=b)
                return temp; // -> Cutoff
            else if (temp>a)
                a=temp; //alpha verbessert
        }
        return a;
    }
}
```

4 Die Hashtabelle – das Gedächtnis eines Schachprogramms

Bisher wurde immer so getan, als ob verschiedene Zugfolgen auch immer zwangsläufig zu verschiedenen Stellungen führen würden – das ist allerdings im Schach und in vielen anderen Spielen nicht der Fall. Zum Beispiel führen, wenn keine Schlagfälle o.ä. dabei sind, Zugfolgen vom Typ **A-B-C** und **C-B-A** zur gleichen Stellung. Es wäre unvorteilhaft, die komplette Suche noch einmal in der Stellung nach **C-B-A** durchzuführen, wenn die Stellung **A-B-C** bereits untersucht wurde. Oftmals kann nämlich das Ergebnis unverändert übernommen werden (die Einschränkung ergibt sich durch evtl. andere alpha- und beta-Werte beim Aufruf!). Es wäre also sinnvoll, wenn das Programm sich merken würde, welche Stellungen es bereits untersucht hat und zu welchem Ergebnis es ggf. gekommen ist. Alle modernen Schachprogramme tun dies auch und speichern diese Ergebnisse in einer sogenannten **„Hashtabelle“**. Es kann allerdings durchaus vorkommen, dass die gleiche Stellung während einer Suche mehrmals bei einer unterschiedlichen Zahl vorangegangener Züge und entsprechend höherer bzw. niedrigerer Resttiefe besichtigt wird. Beispielsweise ist nach Zugfolgen der Form **B-C-B^{-1}-C^{-1}-A** und dem einzelnen Zug **A** die gleiche Stellung entstanden, im ersten Fall aber vorher vier Züge mehr ausgeführt worden. Um zu vermeiden, dass nun nach Ausführung des Zuges A auf das Ergebnis der Suche zu **B-C-B^{-1}-C^{-1}-A** zurückgegriffen wird (denn dort war die verbleibende Resttiefe vier Halbzüge niedriger, das Ergebnis ist dementsprechend unzuverlässiger), muss das Programm sich auch noch merken, welche Tiefe die Suche hatte, auf der der Eintrag basiert (und eigentlich auch noch, ob der gespeicherte Wert exakt war oder lediglich eine obere bzw. untere Schranke für die Partei am Zug darstellt, wie es in der Alpha-Beta-Suche häufig passiert; dieser Aspekt bleibt hier allerdings unberücksichtigt). Stößt es dann während seiner Berechnungen mit Resttiefe **d** auf eine Stellung, zu der zwar ein Hasheintrag existiert, der allerdings auf einer Suche mit Tiefe **d'<d** beruht, so wird die Hashinformation ignoriert; ist dagegen **d'≥d,** so kann der Eintrag ausgewertet werden; bei **d'>d** verbessert sich die Qualität der Suche sogar ein wenig. Eine weitere Verbesserung kann erzielt werden, wenn man sich zusätzlich zu jeder Stellung den besten oder den Zug, der zum Cutoff führte, abspeichert. Diese Information kann man auch im Falle **d'<d** nutzen, denn der beste Zug bei Resttiefe **d'** ist mit hoher Wahrscheinlichkeit auch bei Tiefe **d** noch gut. In diesem Fall unterstützt die Hashtabelle also die Zugsortierung, die ja wie gesehen für eine effiziente Alpha-Beta-Suche sehr wichtig ist. Auf die genaue Implementierung der Hashtabellenfunktion soll hier nicht eingegangen werden; es sei aber erwähnt, dass aus Speicherplatzgründen die Zahl der Hasheinträge (die gelegentlich auch als **„Slots“** bezeichnet werden) begrenzt

werden muss. Da ein modernes Programm in der Regel viel mehr Stellungen besichtigt, als Slots zur Verfügung stehen, müssen häufig bereits bestehende Einträge überschrieben werden, wobei natürlich Information und damit Suchgeschwindigkeit verloren geht. Die Spielstärke eines Schachprogramms hängt also nicht unwesentlich von der Größe des Hauptspeichers ab, der ihm zur Verfügung gestellt wird.

5 Iterative Suche

Will man mit einem Schachprogramm arbeiten oder gar an einem Turnier teilnehmen, so ist es ziemlich ungünstig, wenn zur jeweiligen Brettstellung immer nur eine Suche mit festgelegter Tiefe gestartet und erst nach deren Beendigung das Resultat ausgegeben wird. Oftmals kann ja der Zeitverbrauch hierfür gar nicht zuverlässig vorhergesagt werden, da er sehr stark von der Struktur der gerade am Brett befindlichen Stellung abhängt. Vielmehr sollte das Programm sich sofort eine „Meinung“ bilden und diese dann Schritt für Schritt durch genauere Untersuchung verifizieren oder ggf. ändern; kommt dann während der Partie der Moment, in dem gezogen werden sollte, so wird einfach das Resultat der letzten Untersuchung als Grundlage für die Zugwahl herangezogen. Eine einfache und trotzdem sehr wirkungsvolle Methode ist hierfür die Technik der sog. **„iterativen Suche“**: statt die Brettstellung bereits von Beginn nur mit einer bestimmten Suchtiefe **t** abzusuchen, startet man mit **t=1** und erhöht **t** dann schrittweise jeweils um eins. Jedes mal erfolgt dabei eine gewöhnliche Alpha-Beta-Suche der Tiefe **t**; es hört sich vielleicht widersinnig an, auch die Resultate für niedrigere Tiefen zu berechnen, wenn doch letztlich nur das Ergebnis für die höchste interessant ist. Doch zum einen kann man, wie erwähnt, die im vorgesehenen Zeitrahmen erreichbare Tiefe nicht immer vorhersehen; zum anderen ist es paradoxerweise in der Regel sogar effizienter, die Suchtiefe Stück für Stück zu erhöhen. Dies liegt vor allem in der exponentiell mit **t** wachsenden Zahl an Endknoten und der Möglichkeit zur besseren Zugsortierung begründet: erstere bewirkt, dass der Aufwand für die vorhergehenden Tiefen gegenüber dem für die letzte vergleichsweise gering ist. Zweitere ist gegeben, weil die bei den niedrigeren Tiefen in der Hashtabelle gespeicherte Information, wie im vorherigen Abschnitt beschrieben, für die Zugsortierung verwendet werden kann.

6 Die Ruhesuche

Nun soll eine Lösung zu einem (schachspezifischen) Problem vorgestellt werden, das der bisher vorgestellte Algorithmus mit sich bringt. Würde man tatsächlich immer nur fix bis zu einer festen Tiefe rechnen, so käme es am Ende der Varianten, die das Programm ausgibt, meist zu einem wahren „Gemetzel“. Der Grund: da nach Erreichen der vorgesehen Suchtiefe nicht mehr weitergerechnet wird, werden auch einstehende Figuren nicht mehr erkannt; die Partei, die den letzten Zug ausführen darf, könnte also ungestraft gedeckte Steine schlagen und würde dafür auch noch mit einem hohen Wert belohnt, weil das mögliche Wiederschlagen nicht mehr erfasst wird. Dieser Effekt kann bei längeren Schlagfolgen auch schon ein paar Halbzüge vor Erreichen der Endknoten eintreten. Die auf diese Weise ermittelten besten Züge wären dann oft zufälliger Natur. Grundsätzlich wäre es also besser, wenn die Bewertungsfunktion erst dann aufgerufen wird, wenn “Ruhe“ am Brett herrscht, d.h. fürs erste keine Schlagzüge mehr möglich sind. Da solche Stellungen jedoch oft viel zu spät eintreten, behilft man sich folgendermaßen: ist die entsprechende Tiefe erreicht, so wird der Wert, den die Bewertungsfunktion zurückliefert, als sicherer Wert für die Partei am Zug angesehen. Weiterhin werden nun aber auch noch alle Schlagzüge untersucht; nach Ausführung eines solchen wird wieder der Wert der entstandenen Stellung bestimmt. Diesen hat nun die gegnerische Partei sicher und jetzt werden für diese alle Schlagzüge untersucht und so weiter. Jede Partei kann also, wenn sie am Zug ist, noch eine Schlagserie beginnen oder „aussteigen“. Auf diese Weise wird das oben geschilderte Problem vermieden, denn es ist sichergestellt, dass auf jedes Schlagen das evtl. mögliche Wiederschlagen erkannt wird. Da die Zahl der Schlagzüge in aller Regel sehr gering ist und sich Schlagfolgen auch noch selbst begrenzen (im Schach ist hier nach spätestens dreißig Zügen mangels Material Schluss...), ist der Zusatzaufwand nicht allzu extrem, das Spiel des Programms wird jedoch merklich verbessert.

7 Weniger ist manchmal mehr: der Nullzug

Als Abschluss der Betrachtungen zur Funktionsweise von Schachprogrammen hier noch ein Beispiel für eine, im Gegensatz zur Alpha-Beta-Suche, risikobehaftete Form der Suchbaumreduktion: Schach ist ein Spiel, bei dem es normalerweise immer eine bessere Alternative gibt, als einfach „auszusetzen“ (sprich: dem Gegner das Zugrecht zu überlassen), was die Regeln im übrigen nicht erlauben. Deshalb ist es sehr wahrscheinlich, dass ein Zug, der sich durch die (fiktive) Möglichkeit des Aussetzens widerlegen ließe, auch durch einen regelkonformen Zug als schlecht nachgewiesen werden kann. Darauf basiert die Idee des sog. **„Nullzugs“**, mit dem eben genau jenes Aussetzen bezeichnet wird. Bevor auf einen Zug **z** überhaupt mittels des Zuggenerators alle legalen Gegenzüge ermittelt werden, probiert man, sofern kein Schachgebot vorliegt, erst einmal aus, welche Folgen der

Nullzug hätte und zwar mit einer gewöhnlich um 2-3 Halbzüge reduzierten Tiefe. Wenn **z** ein sinnloser Zug ist oder die gegnerische Stellung gar verschlechtert, so wird er durch den Nullzug widerlegt werden können und aufgrund der reduzierten Tiefe deutlich schneller. Natürlich ist das nicht ganz unproblematisch, könnte es doch durchaus sein, dass eine durch **z** aufgestellte, langzügige Drohung aufgrund der Tiefenreduktion übersehen wird. Da aber mit dem Nullzug auf das eigene Zugrecht verzichtet wurde, stehen die Chancen gut, dass einer der legalen Züge die Drohung pariert hätte und der Cutoff doch zu Recht durchgeführt wurde. Ein Problem sind jedoch die seltenen, aber gerade bei wenig verbliebenem Material immer wieder auftretenden Zugzwangstellungen, in denen eine Seite ihre Stellung mit jedem Zug nur verschlechtern kann. Solche Fälle werden dann ohne entsprechende Verfeinerungen der Nullzugtechnik nicht mehr erkannt, denn die betroffene Partei würde einfach auf das in der Realität nicht mögliche Aussetzen vertrauen. Insgesamt erreicht man mit dem Einsatz des Nullzuges aber deutlich höhere Suchtiefen, was das damit verbundene Risiko mehr als kompensiert.

8 Heutiger Leistungsstand

Fast alle der hier vorgestellten Techniken dürften auch in den derzeit führenden Schachprogrammen eingesetzt werden. Daneben verfügen deren Autoren mit Sicherheit noch über einige Eigenentwicklungen, die Sie jedoch hüten wie Ihren Augapfel. Diese ermöglichen es Ihren Programmen im Zusammenspiel mit der immer schneller werdenden Hardware, selbst absoluten Weltklassespielern Paroli zu bieten, wie eingangs bereits geschrieben. Allerdings soll nicht unerwähnt bleiben, dass diese Leistungsfähigkeit nach wie vor hauptsächlich auf der enormen Rechengewalt und weniger auf einem tieferen Verständnis des Spieles beruht. Immer noch lassen sich problemlos Stellungen konstruieren, in denen jeder durchschnittliche Vereinsspieler besser aussieht als die Maschinen. Offenbar ist es noch niemandem gelungen, ein "intelligentes" Programm zu entwickeln, dass in praktischen Partien vergleichbare Resultate liefern könnte. Auch die in manchen Programmen eingesetzten Mechanismen zum selbstständigen Dazulernen sind nach Kenntnis des Autors bisher nur sehr rudimentär entwickelt und umfassen kaum mehr als die Vermeidung einer zuggenauen Wiederholung von Verlustpartien. Das auf diesem Gebiet aber einiges möglich sein sollte, wird deutlich, wenn man sich vergegenwärtigt, wie ein menschlicher Schachspieler zu seinen Ergebnissen kommt: wenn er gut ist, kann er vielleicht 100-200 Stellungen wirklich berechnen, bevor er seinen Zug wählt. Verglichen mit den zig Millionen, auf die es ein Schachprogramm bringt, fast beschämend wenig. Und dennoch können die besten Menschen immer noch mit den Rechnern mithalten und sogar Partien gegen sie gewinnen. Es wird spannend sein zu beobachten, wie lange dieser Zustand noch anhält... ■

Der Autor (rechts) mit „Jonny" gegen den amtierenden Weltmeister „Shredder" von Stefan Meyer-Kahlen (links) bei der Computer-WM in Graz 2003