Klipp & Klar: Chaos-System Deep Engine

Prozessoren mit mehreren Kernen gehören heute fast schon zum Standard; in wenigen Monaten werden selbst Einsteiger-PCs damit ausgerüstet. Schach-Engines profitieren davon nicht automatisch, aber von vielen Programmen gibt es spezielle Deep-Versionen, die Mehrprozessor-Systeme unterstützen. Die Spielstärke steigt dadurch, das ist unbestritten, aber in Teststellungen liefern die Multi-Engines oft höchst merkwürdige Ergebnisse, sind manchmal sogar langsamer als die Single-Programme, und bei jedem Test gibt es andere Ergebnisse. Dasselbe passiert oft bei unterschiedlichen Hashgrößen. Aber sollten Schachprogramme nicht immer reproduzierbare Resultate liefern? Es sind doch deterministische Systeme, die keinen eingebauten Zufallsgenerator haben! Programmierer sprechen gern von „Quanteneffekten“ – was das genau bedeutet, hat CSS Online untersucht.

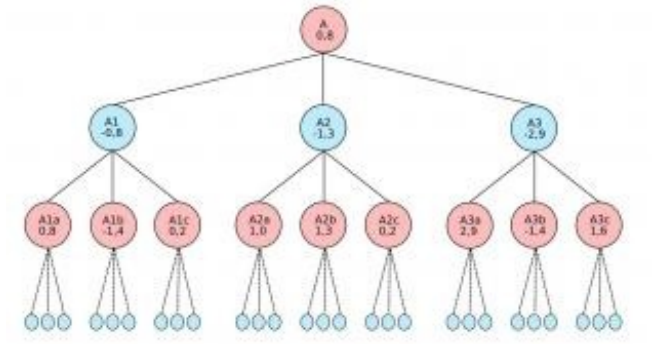
Wir können uns die Sache einfach machen. Ein Programm, das unter Windows oder einem beliebigen anderen Betriebssystem läuft, hat den Rechner nie wirklich für sich allein, sondern muss sich die Rechenzeit mit einem ganzen Rudel anderer Programme teilen. Dienste, Hilfsprogramme, das Betriebssystem selbst, alle können zu beliebiger Zeit dazwischenfunken und tun es auch. Auf einem Single-Prozessor-Computer fällt das nicht ins Gewicht, weil das Schachprogramm einfach wartet und dann da weitermacht, wo es aufgehört hat. Wenn sich aber mehrere Prozessoren mit Schach beschäftigen, wird vielleicht nur einer kurz für andere Aufgaben benötigt, während die anderen weiterrechnen. Weil es immer nur ein paar Millisekunden sind, die das Betriebssystem abknapst, fällt das normalerweise nicht groß auf, aber das Deep-Schachprogramm kann in dieser Zeit eine Menge rechnen, und die Ergebnisse der anderen Prozessoren, die nicht warten mussten, sorgen dafür, dass die Engine letztlich jedesmal einen anderen Suchbaum durchforstet. So erklären sich all die Unterschiede.

Reicht Ihnen das als Erklärung dafür, dass ein paar verlorene Millisekunden oft mehrere Minuten Unterschied in der Lösezeit verursachen? Dann können Sie hier aufhören zu lesen. Alle anderen, die es genauer wissen wollen, seien gewarnt: es wird ein kleines bisschen technisch. Denn um zu verstehen, was genau geschieht, muss man verstehen, wie ein Schachprogramm seine Baumsuche durchführt. Aber eigentlich ist das gar nicht so kompliziert ...

Minimax und Alpha-Beta

Schach ist ein ganz simples Spiel. Ich hierhin, er dorthin, ich hier, er da ... Wenn man dieses Verfahren immer wieder anwendet, ergibt sich die korrekte Bewertung einer beliebigen Stellung von selbst. Schach ist aber auch ein kompliziertes Spiel, zumindest, wenn man nur ein paar Milliarden Jahre Zeit hat – oder noch weniger. Denn der Suchbaum wächst exponentiell, und schon nach wenigen Halbzügen gibt es Fantastillionen von Möglichkeiten. Darum haben kluge Programmierer beschlossen, die Tiefe des Suchbaums zu begrenzen. Damit haben sie sich aber das Problem eingehandelt, dass nun keineswegs mehr die beste Möglichkeit ermittelt werden kann, sondern bloß irgendeine innerhalb der Suchtiefe *noch nicht als suboptimal erkannte Variante* gespielt wird – aber immerhin in endlicher Zeit. Aus diesem Grund gibt es eine Bewertungsfunktion – ein Programmteil, der sich eine Stellung anschaut und versucht, die Chancen des am Zuge befindlichen Spielers in eine Dezimalzahl zu quetschen. Plus 0,88 zum Beispiel, was andeuten soll, dass ein Spieler einen Vorteil hat, der dem Gegenwert von 0,88 Bauern entspricht. Klingt ein bisschen nach einer kleinen Tüte Hobelspäne, funktioniert aber, sofern eine möglichst tiefe Suche vorher dafür sorgt, dass das Programm nicht zweizügig die Tante wegwirft. Chrilly Donninger schrieb mal, die Suche bestimme die Spielstärke, die Bewertung den Stil. Das scheint übertrieben, aber nicht so sehr, wie beispielsweise Fruit und Rybka belegen.

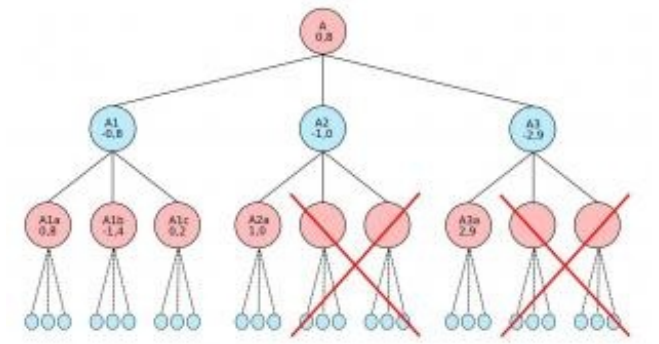
Und genau da liegt der Hase im Pfeffer: Wie funktioniert so eine Suche, und wie wird sie möglichst tief? Das „ich hierhin, er dorthin“ funktioniert so:



Der Einfachheit halber gehen wir hier davon aus, dass auf jeden Zug drei Gegenzüge möglich sind. Eine Engine möchte in einer Stellung wissen, wie gut Zug A (ganz oben im Baum) wirklich ist. Sie erzeugt den Zug A1, den Folgeknoten A1a und vielleicht noch weitere. Die Bewertung von A1a aus Sicht des am Zuge befindlichen Gegners beträgt +0,8 Bauerneinheiten. Ob dieser Wert durch direkte Bewertung oder noch tiefere Suche ermittelt wurde, ist hier nicht wichtig. Der Spieler am Zug hat aber noch die Züge A1b und A1c, die das Programm mit -1,4 und +0,2 bewertet. Von diesen drei Zügen ist A1a der beste, weil +0,8 größer ist als +0,2 und erst recht größer als -1,4. A1a würde also gewählt und an A1 durchgereicht. Dort befindet sich der Gegner am Zug; aus seiner Sicht handelt es sich nicht um +0,8, sondern um -0,8.

Dasselbe geschieht nun mit all den anderen Knoten; am Ende bewertet das Programm A1 mit -0,8, A2 mit -1,3 und A3 mit -2,9. Es erkennt A1 als stärksten (oder wenigstens am besten bewerteten) Zug und A bekommt +0,8 Bauerneinheiten. Plus, weil mit jeder Ebene im Suchbaum das Zugrecht wechselt und alle Züge aus Sicht der Seite am Zuge bewertet werden. Dieses Verfahren heißt Minimax.

Wer Minimax in einem Schachprogramm verwendet, wird bestimmt keinen Ökonomiepreis gewinnen, denn der Suchbaum wächst sehr schnell – mit jedem Halbzug Suchtiefe würde sich die Anzahl der untersuchenden Stellungen im Beispiel verdreifachen. Ein Schachprogramm findet pro Stellung aber nicht drei, sondern ungefähr vierzig mögliche Züge, die es berechnen soll. Den Aufwand zu reduzieren hilft der Alpha-Beta-Algorithmus:



Nachdem das Programm für die Stellung A1 aus den (hier drei) Folgezügen den Wert -0,8 ermittelt hat, beginnt es, A2 zu berechnen. Der erste Folgezug, A2a, liefert als Ergebnis 1,0, A2 würde also zunächst mit -1,0 bewertet werden, bevor A2b und A2c kalkuliert werden. Aber es ist gar nicht mehr nötig, diese beiden Stellungen zu bewerten! Mal angenommen, A2b würde besser bewertet als A2a (was ja im Minimax-Programm der Fall ist), dann würde der Wert von A2 noch kleiner als -1,0. Die -1,0-Bewertung stellen also eine Schranke dar – besser kann es für die Seite am Zug hier nicht mehr werden, nur schlechter. Weil auf dieser Ebene aber der Zug A1 schon mit -0,8 vorliegt, wird Zug A2 ohnehin niemals gewählt, egal, ob er nun mit -1,0 bewertet wird oder mit -2,0. Darum muss das Programm die Knoten A2b und A2c gar nicht erst ansehen.

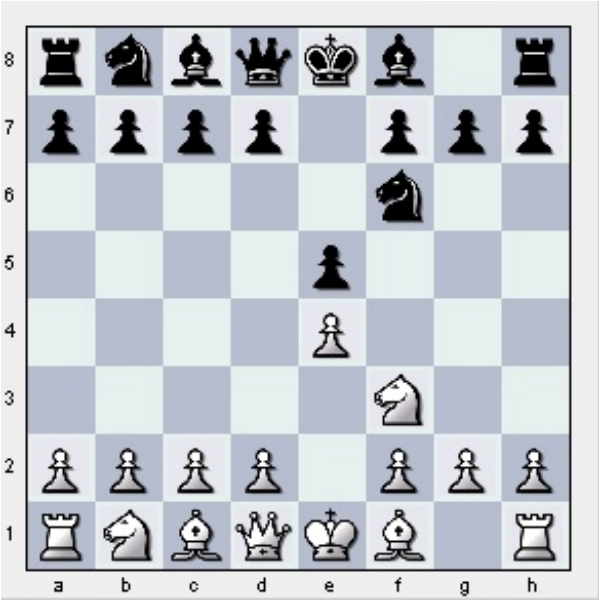
Dasselbe gilt für A3 – auch hier steht bereits nach A3a fest, dass A3 schlechter sein muss als A1, weitere Züge zu untersuchen kann die Sache nicht besser machen!

Der Algorithmus verwendet zwei Schranken, eben Alpha und Beta, welche in der gesamten Suche mitgeführt und laufend angepasst werden. Alpha bezeichnet die untere Schranke der Bewertung; jede Variante, deren Ergebnis schlechter als Alpha ist, taugt nichts, weil der Algorithmus bereits eine Variante kennt, deren Ergebnis mindestens Alpha ist. Beim Start der Suche wird Alpha daher auf den niedrigstmöglichen Wert gesetzt. Beta ist eine obere Schranke – liefert die Suche einen Wert darüber (oder gleich Beta) zurück, bedeutet das, diese Variante würde nie gespielt werden, weil sie für die Seite am Zug so gut ist, dass die Gegenseite sie vermeiden wird – und auch die Möglichkeit dazu hat. In diesem Fall kann die weitere Suche in der Variante abgebrochen werden, denn es ergibt ja keinen Sinn, eine Variante weiterzuspinnen, die der Gegner auf jeden Fall vermeidet.

Trotz der gewaltigen Einsparung – die Anzahl der zu untersuchenden Stellungen wächst mit jeder neuen Suchtiefe nicht mehr in Potenz zur Anzahl der möglichen Züge, sondern nur noch der Quadratwurzel daraus – fehlt dem Alpha-Beta-Algorithmus noch etwas ganz Entscheidendes, nämlich ein Gedächtnis.

Hashtables

Die Anzahl der möglichen Stellungen im Schach beträgt laut Wikipedia etwa 2,28 x 10^46, mögliche Spielverläufe gibt es aber bis zu 10^120. Daraus ergibt sich, dass fast jede Stellung auf mehrere verschiedene Arten, über unterschiedliche Zugfolgen erreichbar sein muss.



Die Grundstellung der russischen Verteidigung kann auf sehr verschiedene Arten auf das Brett kommen:

1.e4 e5 2.Sf3 Sf6

1.e4 Sf6 2.Sf3 e5

1.Sf3 Sf6 2.e4 e5

1.Sf3 e5 2.e4 Sf6

Vier Möglichkeiten, in zwei Zügen dieselbe Position zu erreichen, aber das war noch gar nicht alles, denn die Bauern müssen keinen Doppelschritt ausführen, sondern könnten auch via e2-e3-e4 und e7-e6-e5 auf ihre Zielfelder gelangen, was die Anzahl der Möglichkeiten noch einmal dramatisch erhöht:

1.e3 e6 2.e4 e5 3.Sf3 Sf6

1.e3 e6 2.e4 Sf6 3.Sf3 e5

1.e3 e6 2.Sf3 e5 3.e4 Sf6

1.e3 e6 2.Sf3 Sf6 3.e4 e5

1.e3 Sf6 (vier weitere Zugfolgen)

1.Sf3 e6 (vier weitere Zugfolgen)

1.Sf3 Sf6 (vier weitere Zugfolgen)

In drei Zügen gibt es weitere 16 Möglichkeiten, diese Stellung zu erreichen, insgesamt existieren also 20 Wege, um die Grundstellung der russischen Verteidigung aufzubauen!

Ein Schachprogramm ohne Gedächtnis würde jede dieser Stellungen im Baum erzeugen, dann ganz jungfräulich davor stehen und denken: „oh, interessante Stellung, mal weiterrechnen!“ – 20 Mal, für ein und dieselbe Position.

Um derartige Redundanzen zu reduzieren, kommen Hashtabellen zum Einsatz. Es handelt sich um eine Art riesiges Lager, vollgebaut mit ganz vielen durchnummerierten und abschliessbaren Regalfächern. Die Nummer entspricht der Position, und drin liegt die Bewertung, die Suchtiefe, auf der die Stellung untersucht wurde, die Alpha-Beta-Schranken, der beste Gegenzug und vielleicht noch ein paar weitere Sachen, die für eine Baumsuche spannend sein können. Zu jedem Fach gehört ein spezieller Schlüssel, der zusätzlich zur Regalfach-Nummer die Schachposition symbolisiert.

Technisch funktioniert das so: Jede Stellung wird von zwei Zahlen repräsentiert, den sogenannten Zobrist-Keys, einer Hash-Kennung und einem Hash-Index. Der Index dient der Berechnung der Position, an der ein Knoten in der Hashtabelle gespeichert werden soll (der Regalfach-Nummer); diese ergibt sich aus Hash-Index Modulo Anzahl der

Einträge. Dieses Verfahren hat neben seiner Einfachheit und Schnelligkeit noch den Vorteil, unschlagbar flexibel zu sein, denn die Größe der Hashtabelle bleibt dabei völlig beliebig. Da wesentlich mehr potentielle Suchknoten als mögliche Einträge in der Hashtabelle existieren, geschieht es relativ häufig, dass zwei ganz unterschiedliche Stellungen denselben Index beanspruchen. Um entscheiden zu können, ob es sich um identische oder verschiedene Positionen handelt, vergleicht der Hash-Algorithmus die Hash-Kennungen (die Schlüssel!) der beiden Kontrahenten – nur wenn der Schlüssel zum Schloss bzw. die neu erzeugte Kennung zur in der Hashtabelle gespeicherten Kennung passt, sind die Stellungen identisch.

Es gibt meist auch wesentlich mehr mögliche Knoten als Hash-Kennungen – tritt ein solcher Fall auf, verwechselt das Programm zwei ganz unterschiedliche Stellungen, weil es sie für identisch hält; man spricht von einer Hash-Kollision. Doch die Art der Erzeugung der Hash-Zahlen schließt Kollisionen innerhalb einer gewissen Suchtiefe weitgehend aus: Vor Beginn der Suche wird ein Array in der Größe des Suchraumes angelegt und mit Zufallszahlen gefüllt; auch Hash-Index und Hash-Kennung werden mit zufälligen Werten belegt. Bei jeder Erzeugung eines neuen Knotens werden Hash-Index und -Kennung mit dem korrespondierenden Array-Eintrag XOR-verknüpft. Die XOR-Verknüpfung hat den Vorteil, dass eine erneute Ausführung mit denselben Werten die ursprünglichen Index- und Kennungs-Werte wieder herstellt, was sehr wichtig ist, weil in einer Suche ständig nicht nur Züge ausgeführt, sondern auch zurückgenommen werden. Der Nachteil besteht darin, dass die Adressen wild und absolut zufällig über den gesamten Speicherbereich verteilt werden, den die Hashtabelle belegt. Das bremst zum Beispiel den Prozessorcache dramatisch aus, weil der Prozessor nur ein paar Speicherhäppchen *in der Nähe des vorigen Zugriffs* mit in den Cache einliest. Darin unter anderem liegt begründet, warum sehr kleine Hashtabellen manchmal effektiver erscheinen als große; es passt mehr von ihnen in den Prozessorcache, was den Zugriff extrem beschleunigt, denn Hauptspeicherzugriffe gehen vergleichsweise gemächlich vonstatten.

Die Regalfach-Nummer und den Schließfach-Schlüssel dazu kennt das Programm also, und im Tresor liegt dann die ersehnte Bewertung dieser Stellung. Nur weiß das Programm noch nicht, ob es sie auch verwenden darf, um die Suche abzubrechen – dazu muss auch die Suchtiefe, auf der diese Bewertung ermittelt wurde, mindestens so groß sein wie die avisierte Tiefe der gerade aktuellen Suche. Das muss nicht immer der Fall sein, weil eine Stellung in ganz verschiedenen Zügezahlen erreicht werden kann, selbst eine so simple wie die oben angeführte Russisch-Grundstellung. Auch können in der Hashtabelle noch Informationen aus einem früheren Suchvorgang stehen, vom vorigen Zug etwa, oder vom Pondern. Diese Einträge weisen oft nicht die für einen Suchabbruch erforderliche Tiefe auf, nutzen aber trotzdem, weil sie bei der Zugsortierung helfen – dazu weiter unten mehr.

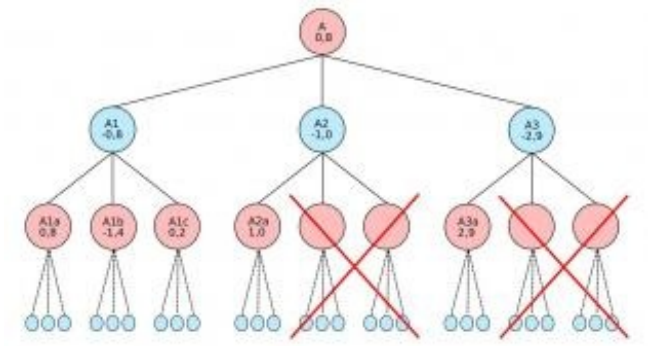
Soll eine Alpha-Beta-Suche gehasht werden, muss ein Hash-Eintrag auch noch Flags enthalten, die anzeigen, ob der gespeicherte Knoten innerhalb des Alpha-Beta-Fensters lag, darüber oder darunter. Da die Werte für Alpha und Beta innerhalb der Suche dynamisch angepasst werden und nur Schranken darstellen, handelt es sich bei der Bewertung eines Knotens nur sehr selten um einen wahren Wert, sondern meist ebenfalls um eine Schranke. In der Hashtabelle muß daher stehen, ob es sich eine obere oder eine untere Schranke handelt. Die Suche darf bei ausreichender Suchtiefe des gespeicherten Eintrags nur abgebrochen werden, wenn ein wahrer Wert gespeichert wurde, der gespeicherte Wert eine obere Schranke und kleiner als das aktuelle Alpha oder der gespeicherte Wert eine untere Schranke und größer als das aktuelle Beta ist.

Adresse	0	1	...	**3550798**	...	Max-1	Max.
Schlüssel	2684987	...	...	**6463217**	...	...	...
Bewertung	-20	...	...	**+122**	...	...	...
Suchtiefe	2	...	...	**3**	...	...	...
Bester Zug	d4	...	...	**Sxe5**	...	...	...
Alpha	-100	...	...	**0**	...	...	...
Beta	+10	...	...	**200**	...	...	...

Soll ein Knoten gespeichert werden, kommt es häufig vor, daß der ermittelte Tabellenplatz bereits besetzt ist. Die Qualität der Entscheidung, welcher der Knoten wichtiger ist und an diesem Tabellenplatz gespeichert werden soll, bestimmt zu einem wesentlichen Teil die Effizienz der Hashtabelle. Die einfachste Replacement-Strategie besteht darin, den gespeicherten Knoten immer zu überschreiben, ausgehend von der Überlegung, dass für den neueren Knoten mehr Rechenzeit verwendet wurde, was meist zutrifft. Ein anderes Verfahren zählt die Knoten, die für die Bewertung der aktuellen Stellung durchsucht werden mußten, und speichert bevorzugt die nach dieser Wichtung besonders wertvollen Knoten. Auch Hashtabellen mit mehreren „Slots“ werden verwendet, um mehr als nur einen Eintrag pro Index speichern zu können. In jedem Fall aber muß sichergestellt werden, dass Knoten nahe der Wurzel nicht von solchen nahe der Blätter überschrieben werden, denn an den Wurzelknoten hängt ein viel größerer Suchbaum, der mit einem Hashtreffer abgeschnitten werden kann, während bei Stellungen nahe an den Blättern eines Suchbaums das Programm nur die geringe Restsuchtiefe einspart.

Zugsortierung

Der durchschnittliche Verzweigungsfaktor im Schach liegt bei 40 Zügen pro Stellung; Alpha-Beta drückt ihn im Idealfall auf ca. 6. Idealfall bedeutet: bei perfekt vorsortierten Zügen. Verfahren, um die Zugsortierung zu verbessern, gehören daher zu den wichtigsten Aufgaben der Schachprogrammierer und sind ziemlich entscheidend für die Effizienz einer Suche.



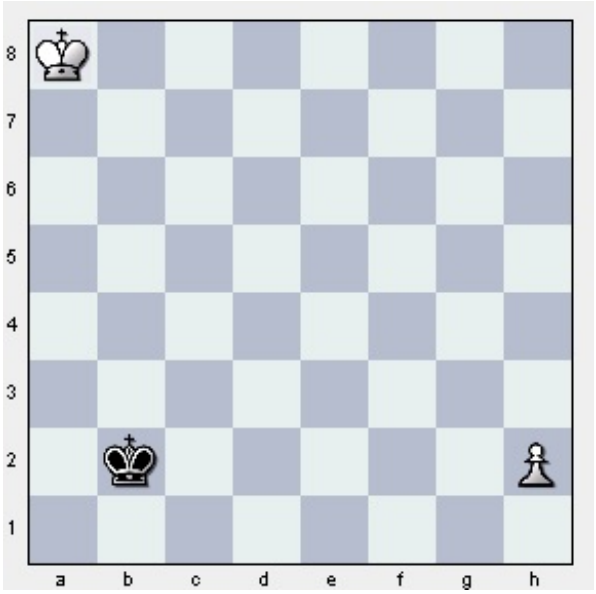
Es ist leicht zu sehen, daß die exakte Einsparung von der Reihenfolge abhängt, in der die Baumknoten durchsucht werden. Ließe man den Alpha-Beta-Algorithmus in Bild nicht den Baum von links nach rechts, sondern von rechts nach links durchsuchen, beginnend also bei Knoten A3c, A3b ..., gäbe es überhaupt keine Einsparung, und die Alpha-Beta-Funktion würde genau soviele Knoten wie Minimax durchlaufen müssen.

Eine Suche kann ganz ohne Wissen über das spezielle Bewertungsproblem Informationen liefern, welche die Zugsortierung drastisch verbessern. Dazu ist es nötig, sich zunächst von dem Depth-first-Konzept zu verabschieden. Obige Beispiele zu den Bildern 1 und 2 gingen davon aus, dass zunächst ein Zweig des Suchbaums von der Wurzel bis zum Blattknoten komplett durchlaufen wird, dann der nächste usw. In der Praxis passiert genau das *nicht*; weil man sich beispielsweise die Chance nimmt, die Suche nach einer bestimmten Zeit abzubrechen und trotzdem ein der Dauer der bisherigen Suche angemessenes Resultat zu bekommen – aussagekräftige Ergebnisse kann die Depth-First-Suche nur liefern, wenn sie komplett durchlaufen wurde. Zudem ist es im Vorfeld meist sehr schwierig abzuschätzen, wie lange eine Suche bestimmter Tiefe dauern wird.

Beim *Iterative Deepening* werden die Baumebenen in aufsteigender Reihe untersucht. In Bild 1 würde der Algorithmus zunächst die Knoten A1, A2 und A3 erzeugen, bewerten und daraus die Bewertung für den Wurzelknoten bestimmen. Erst danach würde der komplette dargestellte Suchbaum durchlaufen werden bis zur Ebene A1a...A3c. Das wirkt zunächst ineffizient, schließlich entsteht dadurch viel Redundanz – die Knoten A1, A2 und A3 werden nun zwei- statt wie bisher einmal durchlaufen, und bei jeder weiteren (nicht dargestellten) Suchtiefenerhöhung erneut, überhaupt werden die Knoten im Innern des Baums mehrfach von der Suche erzeugt. Doch angesichts des exponentiellen Wachstums des Suchbaums liegt die Anzahl der redundant untersuchten Knoten immer deutlich unter der Anzahl der mit einer neuen Suchtiefe hinzukommenden Blattknoten; der zusätzliche Aufwand ist also relativ gering, zumal die Hashtabelle die meisten wiederholt erzeugten Stellungen erschlägt. Muss die Suche durch äußere Umstände abgebrochen werden, bevor die letzte und tiefste Ebene durchsucht wurde, stehen immerhin die Ergebnisse der vorigen Tiefe zur Verfügung.

Tatsächlich beschleunigt *Iterative Deepening* die Suche drastisch, trotz der paar zusätzlich untersuchten Knoten, denn das Verfahren erlaubt eine bessere Zugsortierung und garantiert damit die Effizienz des Alpha-Beta-Algorithmus. Im Bild hängt, wie oben gezeigt, die Anzahl der Beta-Schnitte nur von der Reihenfolge der untersuchten Varianten ab – beginnt die Suche mit A1a und arbeitet sich bis A3c durch, können vier Knoten, der Maximalwert für den dargestellten Suchbaum, abgeschnitten werden. Beginnt die Suche zufällig auf der anderen Seite des Baums, mit A3 und Folgeknoten, gibt es überhaupt keine Beta-Schnitte und alle Knoten müssen wie bei Minimax durchsucht werden. Durch das I*terative Deepening* wüsste man schon vorher in etwa über die Bewertungs-Verhältnisse Bescheid und kann die Knoten minderer Suchtiefe entsprechend sortieren, sodass mutmaßlich gute Varianten zuerst untersucht werden.

Das bekannteste Verfahren, sich gute Züge zu merken, heißt *Killer Heuristik* . In einer Tabelle notiert das Programm für jede Suchtiefe den Zug, der als letzter einen Beta-Cutoff verursacht und die Suche in diesem Zweig damit beendet hat. Weil ein Schachprogramm sich die allermeiste Zeit damit beschäftigt, völlig idiotische Züge intern auszuführen und zu widerlegen, geschieht das recht häufig. Die oft zutreffende Annahme ist, dass ein Zug, der eine Variante widerlegt, das auch mit einer anderen schafft, die im Suchbaum nicht weit weg liegt und darum ähnliche Stellungsbilder beinhaltet.



In dieser Stellung mit Schwarz am Zug könnte das Schachprogramm zuerst 1...Kc2 berechnen. Im Zuge seiner weiteren Kalkulationen würde es tiefer im Baum bemerken, dass der Zug 2.h4 darauf gewinnt, weil Schwarz die Umwandlung nicht mehr verhindern kann. h4 würde zum Killerzug erklärt und fortan auf jeden anderen schwarzen Zug bevorzugt ausprobiert. Wenn die Engine als zweite Variante beispielsweise 1...Ka3 berechnen wollte, müsste sie sich nicht bis tief in den Baum mit den Folgen weißer Königszüge herumschlagen, sondern würde schnell bemerken, dass der vorher gespeicherte Killerzug 2.h4 auch hier gut ist und gewinnt. Damit spart das Programm Zeit und hat die Chance, den einzigen Remis-Zug 1...Kc3 wesentlich schneller zu finden. Natürlich probiert es den Killerzug auch nach Kc3 aus, aber hier gewinnt er nicht mehr. Das Programm gewinnt aber, nämlich Zeit und Erkenntnis!

Die meisten Schachprogramme notieren sich pro Suchtiefe mehrere Killerzüge, die sie nacheinander bevorzugt ausprobieren. Ein anderes Verfahren, *History Heuristik* genannt, führt für alle auf einer Suchtiefe möglichen Züge eine Liste, in die das Programm einträgt, wie oft der betreffende Zug schon einen Beta-Cutoff verursacht hat. Es probiert dann zuerst Züge aus, die schon öfter geschafft haben, einen Zweig abzuschneiden.

Genau an dieser Stelle hilft auch die Hashtabelle weiter. Wann immer das Schachprogramm eine Stellungsbewertung in der Hashtabelle ablegt, notiert es auch den besten Zug. Trifft es im Suchbaum erneut auf diese Stellung, die gespeicherte Suchtiefe reicht aber nicht, um die Suche abzubrechen, oder die Schranken passen nicht zum aktuellen Alpha-Beta-Fenster, weiß es doch immerhin den Zug, der in dieser Stellung schon einmal der beste war. Mit hoher Wahrscheinlichkeit ist er auch unter veränderten Bedingungen gut. Der Hashzug ist das wichtigste Mittel der Zugsortierung, denn er beruht immer auf einer schon durchgeführten Suche. Darum wird er immer zuerst ausprobiert.

All diese Verfahren haben mit Schach überhaupt nichts zu tun, sondern funktionieren in Dame- oder Reversi-Programmen ebenso gut. Man kann für die Zugsortierung aber auch Wissen einsetzen, wobei es hierbei praktisch nie um positionelle Feinheiten geht, sondern einfach darum, Züge bevorzugt auszuprobieren, bei denen eine schwächere eine stärkere Figur schlägt.

Fenstertechnik

Der Alpha-Beta-Algorithmus gibt noch mehr Milch, wenn man ihn kitzelt: Je enger die Schranken für Alpha und Beta zusammenrücken, desto öfter gibt es Suchabbrüche, desto schneller liefert die Suche also ein Ergebnis. Weil die Suche schrittweise vertieft wird, das Programm also zuerst Tiefe 1 komplett berechnet, dann Tiefe 2 usw., kennt es immer schon ungefähr den zu erwartenden Wert des besten Zuges – die Annahme, dass ein Zug, der auf Tiefe X am besten ist, es auch auf Tiefe X+1 bleibt, stimmt meist.

Im Suchbaum passt der Alpha-Beta-Algorithmus seine Schranken dynamisch an, an der Wurzel aber muss man ihm feste Werte vorgeben. Die sichere Variante ist, Alpha auf die schlechtest mögliche und Beta auf die bestmögliche Bewertung zu setzen. Allerdings gäbe ein schmalerer Bereich, das sogenannte Alpha-Beta-Fenster, bessere Cutoffs im Suchbaum, was die Suche beschleunigt. Durch das iterative Vordringen in den Suchbaum ist der an der Wurzel beste Zug für jede bereits berechnete Suchtiefe schon bekannt. Meist ist das auch auf der nächsthöheren Tiefe der beste Zug, darum wird er als erster untersucht. Wenn diese Annahme aber meist zutrifft, ist es gar nicht nötig, die anderen Züge mit einem vollen Alpha-Beta-Fenster zu untersuchen, denn die anderen Züge sind mit großer Wahrscheinlichkeit schlechter. Darum wird nur der (wahrscheinlich) beste Zug mit vollem Alpha-Beta-Fenster untersucht, alle weiteren Züge mit einem Nullfenster: Alpha wird auf den Wert des besten Zuges gesetzt, Beta auf Alpha +1.

Die Suche hat nun keine Chance, einen wahren Wert innerhalb des Alpha-Beta-Fensters zurückzuliefern, sie zeigt aber an, ob der Zug besser oder schlechter ist. Liefert die Suche einen Wert kleiner Alpha, dann bedeutet das, der untersuchte Zug war schlechter als der bisherige beste. In diesem Falle hat sich die Annahme bestätigt und der nächste Zug kann überprüft werden, denn es ist ganz uninteressant, wieviel schlechter genau der Zug ist. Liefert die Suche einen Wert größer Beta (Fail-High), bedeutet das, der untersuchte Zug ist besser als der bisher beste Zug. Wieviel besser, ist nicht bekannt, sondern muss durch eine neue Suche ermittelt werden.

Diese neue Suche muss aber auch nicht mit einem maximal geöffnetem Fenster durchgeführt werden, denn die untere Schranke ist bereits bekannt: der vorher beste Wert. Auch stehen für diese wiederholte Suche schon viele Einträge in der Hashtabelle, die zwar wegen des geänderten Alpha-Beta-Fensters nur selten einen Suchabbruch bewirken, durch die gespeicherten besten Züge aber dennoch die Suche beschleunigen. Unter dem Strich ist die Nullfenster-Suche viel schneller als Alpha-Beta mit offenem Fenster.

Aspiration Window Search baut die Idee der Nullfenstersuche weiter aus. Beim *Iterative Deepening* ist an der Wurzel nicht nur der bisher beste Zug, sondern natürlich auch seine Bewertung bekannt. Wenn die Annahme, dies sei auch auf der nächsthöheren Suchtiefe der wahrscheinlich beste Zug, meistens zutrifft, und das tut sie, könnte doch auch die Bewertung ähnlich sein. Darum wird nur ganz zu Anfang, auf Suchtiefe Eins, mit offenem Fenster gerechnet. Auf allen höheren Suchtiefen sucht der Algorithmus mit der unteren Schranke Alpha = bester Wert - Epsilon und der oberen Schranke Beta = bester Wert + Epsilon. Der exakte Wert von Epsilon hängt von der Art der Bewertungsfunktion ab, insbesondere von deren Bewertungsdifferenzen; ein Bauer plus und ein Bauer minus wäre zum Beispiel ein praktikabler Wert.

Beim Einsatz von *Aspiration Window* muss sowohl Fail-Low als auch Fail-High korrekt zu einem wahren Wert aufgelöst werden, denn es ist wichtig, dass der beste Zug eine absolute, also korrekte Bewertung hat. Bei einem Fail-Low erfolgt daher eine Wiederholung der Suche mit nach unten offenem Fenster; als obere Schranke dient Alpha aus der vorigen Suche, also die bisher untere Schranke, weil der wahre Wert auf jeden Fall darunter liegen muss. Liefert die Aspiration-Search ein Fail-High, wird die bisher obere Schranke als untere verwendet und mit einem nach oben geöffneten Fenster die Suche wiederholt. *Aspiration Window* wirkt sich ausschließlich auf den besten Zug an der Wurzel aus; alle anderen Züge werden wie oben beschrieben mit einem Nullfenster gerechnet.

Beachtenswert dabei ist, dass die höhere Effizienz, die durch die Verkleinerung des Suchfensters erreicht wird, durch mehrere Wiederholungssuchen erkauft wird. Aber einerseits beschleunigt eine vorhandene Hashtabelle diese Suche schon sehr, weil viele Positionen bereits bewertet wurden und daher zumindest der beste Zug in vielen Stellungen schon bekannt ist, andererseits erfolgen auch diese Wiederholungssuchen mit Fenstern, die nur in eine Richtung geöffnet wurden. Eine Verfeinerung dieser Idee, welche die meisten Schachprogramme auch verwenden, besteht darin, das Fenster nicht komplett in eine Richtung zu öffnen, sondern nur einen bestimmten Bereich. Das beschleunigt die Wiederholungs-Suche, birgt aber das Risiko eines Resultats, das erneut aus dem Fenster fällt und eine weitere Wiederholung erfordert.

Die Idee der Nullfenster-Suche muss sich nicht auf die Wurzel des Suchbaumes beschränken, sie kann leicht abgewandelt auch im Suchbaum verwendet werden und heißt dann *Principle Variation Search*, PVS.

Doch diese Ader trägt noch mehr Gold. Was, wenn man von vornherein jeden Zug mit einem Nullfenster rechnet? Schneller kann die Suche dann nicht mehr werden, freilich liefert sie jedesmal ein Fail-High oder Fail-Low zurück. Die Suchtechnik *MTD(f)* versucht, durch fortgesetzte Nullfenster-Suche den wahren Wert zu ermitteln. Dazu macht *MTD(f)* zunächst eine erste Schätzung des wahren Wertes und sucht mit dieser Schätzung als Schranke. Je nachdem, ob die Suche einen Wert über oder unter der Schranke zurückliefert, paßt *MTD(f)* die Schranke nach oben oder unten an und sucht erneut, solange, bis die Schranke gegen den wahren Wert konvergiert. Theoretisch kann *MTD(f)* so bessere Ergebnisse als andere Suchverfahren erreichen, praktisch gibt es kaum Unterschiede zu *Aspiration Window Search* mit PVS.

Selektive Suche

Alpha-Beta mit allen beschriebenen Erweiterungen und Optimierungen krankt daran, dass die Suchtiefe pro Iteration fix ist. Alles innerhalb der Suchtiefe wird perfekt behandelt, alles außerhalb dieser Suchtiefe liegt im Zwielicht einer meist höchst approximativen Bewertung. Es gibt eine saubere Trennung zwischen Perfektion und Blindheit an den Blättern des Suchbaums. Daraus resultierende Probleme heißen *Horizonteffekt*; sie entstehen, weil eine Suche mit konstanter Tiefe oft unangemessen ist, denn in verschiedenen Ästen des Suchbaums können sich Lösungen auf unterschiedlicher Tiefe ergeben. Darum setzen baumsuchende Programme oft Techniken ein, um sinnvoll erscheinende Varianten tiefer zu verfolgen, als die nominelle Suchtiefe es zuließe, und offensichtlich unsinnige Varianten vor Erreichen der maximalen Suchtiefe zu beenden. Damit verlassen sie allerdings die sicheren Gewässer der korrekten Suche und segeln aufs stürmische Meer der Spekulation hinaus – waren bei den bisher beschriebenen Techniken der beste Zug und seine Bewertung identisch mit den Werten einer Minimax-Suche, so gilt das bei Einsatz selektiver Techniken nicht mehr unbedingt.

Verfahren zur Vertiefung der Suche wurden bereits von Shannon vorgeschlagen. Shannon wollte die Suche solange fortsetzen, bis „ruhige“ Stellungen entstanden sind – im Schach, auf das er sich bezog, also keine Figuren geschlagen werden können. Das erlaubt nicht nur, die Bewertung simpler zu halten, sondern mindert auch den Horizonteffekt. Ruhesuche nach bestimmten bewertungsspezifischen Termen gehört fest zu jedem Programm, das eine Baumsuche einsetzt.

Grob kann man selektive Suche nach wissensspezifischen und suchspezifischen Verfahren unterscheiden. Wissen bedeutet in diesem Zusammenhang vor allem, die Suche bei Schlagzügen oder Schachgeboten zu vertiefen, es existieren jedoch auch Konzepte, besonders interessante Bewertungsmerkmale zum Maßstab zu nehmen; ein Freibauernpaar vielleicht, oder eine bestimmte Konstellation beim Angriff auf einen König. Solche Varianten werden dann tiefer untersucht. Es gibt derartige statische (stellungsabhängige) Erweiterungen in den meisten Schachprogrammen, und die Idee scheint auch sehr logisch, praktisch dominieren jedoch die dynamischen, von der Stellung unabhängigen Erweiterungen.

Das Deep-Blue-Team, das durch den Sieg ihres Computers über Schachweltmeister Kasparow bekannt wurde, beschrieb *Singular Extensions* : Wenn in einer Stellung nur ein einziger guter Zug existiert, dann soll diese Variante eine Iteration tiefer untersucht werden als vorgesehen. Es ist ein kleines Problem, überhaupt herauszufinden, ob ein Zug singulär ist, denn genau diese Information, den Bewertungsabstand des besten zum zweitbesten Zug, liefert der Alpha-Beta-Algorithmus im Unterschied zu Minimax gerade nicht mehr. Es wurden aber Verfahren entdeckt, um die mögliche Singularität eines Zuges hinreichend genau schätzen zu können. Zum Beispiel könnte ein Schachprogramm die anderen Züge mit einem etwas weiter geöffneten Fenster und deutlich reduzierter Suchtiefe mal kurz „anrechnen“. Das kostet zwar trotzdem Zeit, und in manchen Stellungen würde eine Lösung dadurch später gefunden, die Hoffnung der Singular-Extension-Verwender ist aber, dass die Ergebnisse der Suchbaum-Vertiefung den Mehraufwand meist aufwiegen. Singular Extensions haben mit dem tatsächlichen Problem nichts zu tun, sie funktionieren in jeder Baumsuche, auch in Dame- oder Mühle-Programmen, unabhängig von der Bewertung. Der Vorteil von Singular Extensions besteht darin, dass die Suche auch bei so genannten stillen Zügen, die nichts schlagen und kein Schach geben, vertieft wird.

Schwieriger als Varianten, die vertieft untersucht werden können, sind welche zu finden, die vor Erreichen der nominellen Suchtiefe abgebrochen werden können. Wissensbasierte Verfahren fristen hier erst recht ein Mauerblümchendasein; es dominieren die suchabhängigen Methoden. Ein Beispiel ist die *Nullmove-Heuristik* . Dabei geht man davon aus, dass es normalerweise keine gute Idee ist, gar nichts zu tun. In einer Alpha-Beta-Suche führt die *Nullmove-Heuristik* als ersten Zug einen Nullzug aus, der die Stellung nicht verändert, und sucht danach ganz normal weiter, allerdings mit verkürzter Suchtiefe. Gelingt es der Gegenseite nicht, vom Nullzug zu profitieren, so taugt mit hoher Wahrscheinlichkeit schon die gesamte Variante nichts; der Algorithmus bricht die Suche ab und wendet sich erfolgversprechenderen Fortsetzungen zu. Das Nullsuch-Paradies beherbergt aber auch eine fiese Schlange, denn leider wäre es im Schach manchmal von Vorteil, nicht ziehen zu müssen – besonders im Endspiel existieren viele Zugzwang-Motive, an denen Nullsucher schmählich scheitern. Die Verfahren, Zugzwang zu erkennen, sind zwangsläufig höchst primitiv, weil sie ja nicht den Zeitgewinn auffressen dürfen, den die Nullmove-Technik bringt, wenn *kein* Zugzwang vorliegt.

Parallelisierung

Was das Ganze nun mit Parallelisierung zu tun hat, mit den dramatischen Unterschieden zwischen Lösezeiten unter scheinbar gleichen Bedingungen? Ganz einfach! Eine Aufteilung des Suchbaums auf mehrere Prozessoren ist nur bei reinem Minimax trivial; das Ganze zu parallelisieren macht nicht mehr Mühe als eine Packung Snickers aufzureißen. Bei drei Prozessoren würde einer den Zug A1, der nächste den Zug A2 und der letzte A3 berechnen. In gewissen Grenzen entstünde nicht einmal Verlust, das Programm würde mit zunehmender Prozessor-Anzahl proportional mehr suchen, und in jedem Falle verhielte es sich deterministisch, würde also mit jedem Durchlauf für eine Stellung dasselbe Ergebnis auswerfen und etwa dieselbe Zeit brauchen.

Die Probleme treten erst mit Alpha-Beta auf – die Schranken ändern sich überall im Baum auf nicht vorhersehbare Weise. Wenn eine Stellung von einem Thread bewertet und in die Hashtabelle geschrieben wird, kann ein anderer Thread sie daraus lesen, sobald er auf anderem Wege dieselbe Stellung erreicht hat. Nur wird bei beiden Threads das Alpha-Beta-Fenster ein ganz anderes sein. Wenn jetzt ein Thread geringfügig verzögert wird, erreicht ein anderer Thread die Stellung zuerst , sucht sie vergebens in der Hashtabelle und schreibt seinen ermittelten Wert hinein. Wegen der unterschiedlichen Alpha-Beta-Schranken hat das signifikanten Einfluss auf den zu durchlaufenden Suchbaum, denn andere Schranken produzieren andere Cutoffs. Die ermittelte Bewertung sollte bei reinem parallelisiertem Alpha-Beta praktisch identisch sein, für die Lösezeit muss das aber keineswegs gelten.

Verkompliziert wird das Ganze durch Suchbaum-Beschneidungen wie Nullmove oder Late-Move-Reductions und durch Erweiterungen wie Singular Extensions. All diese hängen direkt vom durchlaufenen Suchbaum ab; ein anderer Suchbaum triggert andere Erweiterungen, andere Abschneidungen. Viele besonders gute Züge findet ein Schachprogramm aber zuerst in vertieft untersuchten Varianten. Einmal nach vorn sortiert, ist es eine Kleinigkeit, sie bis zum Ende zu durchsuchen, das Problem besteht darin, sie erstmal für interessant zu befinden. Ob und wann das geschieht, hängt in einem parallel arbeitenden Programm ausschließlich vom Zufall ab.

Ein gutes Beispiel sind die Stellungstests: da geht es nicht um ganz normale, sondern um außergewöhnliche Züge, um Züge, die nicht auf der Hand liegen für ein Schachprogramm. Für die meisten dieser Stellungstest-Probleme gibt es auch kein Spezialwissen; die Lösezeit hängt also einzig davon ab, wann ein Programm in einer Sucherweiterung auf die korrekte Zugfolge stößt und der Einleitungszug nach vorn sortiert wird. Das geschieht rein zufällig, und je mehr Prozessoren, desto zufälliger und weniger reproduzierbar wird die Geschichte, denn schon die Verzögerung eines Threads um ein paar Millisekunden bewirkt die fantastischsten Veränderungen im Suchbaum.

Dasselbe Problem in abgeschwächter Form tritt übrigens auch auf, wenn man ein Programm mit verschiedenen Hashgrößen auf eine Stellung loslässt. Denn dann müssen bei kleinerer Hashtabelle öfter Stellungen überschrieben werden. Das Resultat bleibt dasselbe wie bei der Parallelisierung: Der untersuchte Baum ist ein anderer, und die Lösezeiten können drastisch voneinander abweichen, umso mehr, je stärker das betreffende Sucherweiterungen benutzt.

Faktor X

Die Hersteller überschlagen sich, wenn es darum geht, die Effizienz der Parallelisierung zu loben, üblicherweise bekommt man einen Steigerungsfaktor um die Ohren geschlagen – mit zwei Prozessoren um Faktor 1,9 schneller, mit vieren 3,7 usw. – leider versäumen die rührigen Programmierer, den Käufern mitzuteilen, was „schneller“ in diesem Zusammenhang genau bedeutet. Knotenzahlen gehören hier wie so oft zu den eher ungeeigneten Messgrößen. Man stelle sich mal die einfachste Form der Parallelisierung auf zwei Prozessoren vor: an einer beliebigen Stelle im Baum wird die vorsortierte Zugliste in der Mitte geteilt, ein Prozessor berechnet die obere Hälfte, der andere die untere Hälfte. Weil Alpha-Beta nur mit guter Zugsortierung vernünftig funktioniert und darum eben dafür ein riesiger Aufwand stattfindet, wird fast immer einer der Züge der oberen Hälte, ob Hash-, Killer- oder Schlagzug, am besten sein und ausreichen, die Variante abzuschließen. Die zweite Hälfte der Zugliste würde auch von einem Single-Programm fast nie berechnet werden.

Ein auf diese Weise parallelisiertes Programm würde locker die doppelte Knotenanzahl anzeigen, aber es würde kein bisschen besser spielen. Möglich, dass in vereinzelten Teststellungen ein zuerst für schlecht gehaltener Zug, der durch den zweiten Prozessor trotzdem berechnet würde, zu fantastischen „Steigerungen“ führte, aber im allgemeinen würde auch in Testsuites kaum eine Verbesserung erkennbar sein – trotz doppelter Knotenzahl.

Kein Programmierer wird natürlich eine so dämliche Parallelisierung verwenden, sondern seine Züge cleverer auf mehrere Threads aufteilen. Trotzdem bleibt es zwangsläufig so, dass verschiedene Prozessoren, die an verschiedenen Suchbäumen basteln, öfter mal auf Stellungen stoßen, die ein anderer Prozessor schon berechnet hat. Deren Ergebnis können sie dann ohne eigene Berechnung aus der Hashtabelle lesen, schreiben sich dafür aber ratzfix eine berechnete Stellung an. Das treibt natürlich die Knotenzahl in die Höhe, obwohl nur schon Getanes nochmal gezählt wurde. Es hilft aber trotzdem, weil die einzelnen Threads ja miteinander kommunizieren *müssen*, um ihre Ergebnisse auszutauschen.

Amir Ban und Shay Bushinsky

Der einfachste Weg besteht darin, diese Kommunikation ausschließlich über die Hashtabelle stattfinden zu lassen. Da hat man nicht viel Stress, muss nur den Suchbaum an geeigneter Stelle aufteilen, und der Rest funktioniert von selbst. Es funktioniert aber nur, wenn alle Prozessoren auf dieselbe Hashtabelle zugreifen können, die Maschine also shared memory hat. Das Verfahren hat einen Nachteil: es skaliert schlecht; je mehr Prozessoren, desto schlechter funktioniert die Sache. Das liegt daran, dass immer nur ein Thread seine Ergebnisse in die Hashtabelle schreiben kann. Die anderen müssen solange warten. Je mehr Prozessoren nun ihre Resultate im Speicher verewigen möchten, desto größer wird das Skalierungs-Problem. Beim Schaukampf zwischen DeepJunior und Kasparow hatte das Junior-Team zwei Maschinen, eine mit vier und eine geringfügig niedriger getaktete mit acht Prozessoren. Die Programmierer Bushinsky und Ban erzählten dem Sponsor zuliebe allen, sie spielten mit der Achtprozessor-Maschine, aber sie taten es nicht, sondern griffen auf das kleinere System zurück – ein Skalierungsproblem.

Andere Programme benutzen bessere, aber auch aufwendigere Verfahren, und wenn kein gemeinsamer Speicher zur Verfügung steht, dann bleibt ihnen auch nichts anderes übrig. Bei Hydra zum Beispiel kommunizieren alle Rechenthreads miteinander. Wenn einer gerade fertig mit seinem Stück vom Suchbaum ist, schickt er an die anderen eine Nachricht: „Hey, ich hab‘ Langweile, hat einer was zu tun?“, und einer ist immer dabei, der ein bisschen Arbeit abgeben möchte, sagte Ex-Hydra-Parallelisierer Dr. Ulf Lorenz gegenüber CSS. Dabei hängt natürlich alles an der Geschwindigkeit dieser Kommunikation, darum wurden die einzelnen Hydra-Rechner mit extrem schnellen (und extrem teuren) Myrinet-Karten ausgerüstet.

Angesichts all der Unwägbarkeiten einer parallelen Suche gibt es eigentlich kein wirklich zuverlässiges Mittel, um die Effizienz der Parallelisierung exaxt zu messen. Am plausibelsten ist es laut Ulf Lorenz, die Zeit zu messen, die das Programm benötigt, um eine


Der Hydra-Cluster (in der Mitte)

Myrinet 10-GBit-Karte

bestimmte Suchtiefe zu erreichen – und zwar mehrmals, um Abweichungen erfassen und herausmitteln zu können. Konkrete Versuche laufen bereits im CSS-Labor; Timo Klaustermeyer wird demnächst seine Messungen präsentieren.

Fazit

Das ganze Problem ist eigentlich keins und resultiert aus der unterschiedlichen Denkweise von Menschen und Computerprogrammen. Ein Mensch pauschalisiert, er kann die Ergebnisse einer Variante auf andere, ähnliche Varianten übertragen. Unbewusst erwartet er das auch von seiner Engine: „Das *muss* sie doch sehen, eben hat sie doch auch, und es ging ganz schnell, wieso jetzt nicht mehr?“

Schachprogramme rechnen aber bloß, darum müssen sie gar nichts sehen und tun es auch nicht. Um einen einzigen Zug auszuführen, erzeugen und bewerten sie viele Millionen Stellungen, die zusammen einen gigantischen Suchbaum ergeben – ich hierhin, er dorthin ... All die raffinierten Tricks in der Suche (und in diesem Artikel wurden nur die trivialsten dargestellt) bewirken vor allem, dass es von absolut entscheidender Bedeutung für die Lösezeit ist, wann genau ein Zug für interessant genug erachtet wird, um nicht abgeschnitten, sondern weit vorn in der Zugliste einsortiert zu werden. Das wiederum hängt entscheidend an den Sucherweiterungen, und wann da welche anspringt mit welchem Resultat, das hängt vom vorher untersuchten Baum ab.

Der Suchbaum selbst ist aber ein fragiles Geschöpf, dessen genaue Gestalt von vielerlei Faktoren abhängt, unter anderem von der Größe der Hashtabelle (und ihrem Replacement-Algorithmus), von der Reihenfolge, in der verschiedene Threads ihre Resultate in die Hashtabelle schreiben, vom Parallelisierungsalgorithmus selbst und von etlichen anderen Dingen. Mit diesem Nichtdeterminismus muss man leben; er hat ja auch Vorteile: dass mit der immer größeren Verbreitung von Multiprozessor-Maschinen und dafür geeigneten Programmen den Stellungstest endlich der Platz angewiesen wird, den sie verdienen. Den eines netten und interessanten, aber nicht besonders aussagekräftigen Spielzeugs nämlich. *(Lars Bremer)*