Controlled Conspiracy Number Search

Schriftliche Arbeit zur Erlangung
des Doktorgrades
des Fachbereichs Mathematik / Informatik
der Universität-Gesamthochschule Paderborn

von

Ulf Lorenz

Fachbereich Mathematik / Informatik
Universität GH Paderborn
Paderborn, Deutschland

Dezember 2000

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Spiele

Jeder von uns steht immer wieder einmal vor der Aufgabe, eine Folge von Entscheidungen innerhalb einer beschränkten Zeitspanne treffen zu müssen. Normalerweise haben wir dabei nicht genügend Zeit, alle Konsequenzen unserer Entscheidung zu berücksichtigen. Einmal gefällt, ist unsere Entscheidung unwiderruflich. Als besonders unangenehm empfinden wir dabei Situationen, in denen es andere Personen gibt, die aus unseren Fehlern auf unsere Kosten einen Gewinn für sich selber erzielen wollen. Solche Situationen möchte man besser verstehen und darin nach Möglichkeit sogar Computer zu konkurrenzfähigen Akteuren machen.

Nun hätte selbst Newton wohl kaum die Bewegungsgesetze entdeckt, wenn er versucht hätte, Wasserfälle und Wirbelstürme zu verstehen. Stattdessen hat er das Problem auf den ursprünglichsten Fall, den er sich vorstellen konnte, vereinfacht: auf Planeten, die sich im Vakuum bewegen. Typische Vorgehensweisen bei der Modellierung komplexer Zusammenhänge sind also Abstraktion und Vereinfachung.

Modelle, die sich mit der zu Anfang beschriebenen Art von Problemen beschäftigen, bezeichnen wir als Spiele. Man findet sie in Form von Gesellschafts- und Lernspielen, wie z.B. Schach, Dame oder Othello, oder als Modelle von komplexen wirtschaftlichen Zusammenhängen [vNM43]. Spiele sind somit selber kleine Welten, die es erlauben, sich auf das Wesentliche dessen zu konzentrieren, was eine vorausschauende Entscheidung ausmacht.

Sie bilden kleine Welten, die durch wenige, einfache Regeln beschrieben werden können. Spiele stellen klare Kriterien für Erfolg und Mißerfolg zur Verfügung, erfordern kein allzu großes Faktenwissen und sind obendrein einfach zu programmieren. Deshalb sind sie

schon immer als ideale Testumgebungen Teil der Künstlichen Intelligenz und als Untersuchungsgegenstand Teil der Mathematik [EBG82] gewesen. Man kann eine Reihe von interessanten und natürlichen mathematischen Problemen, die zugleich die schwierigsten in den Komplexitätsklassen PSPACE, EXPTIME oder EXPSPACE sind, als Zweipersonenspiele auffassen[Fra95].

Eine Methode, um in Zweipersonenspielen gute Entscheidungen zu fällen, besteht darin, für alle möglichen Situationen, die auftreten können, die bestmögliche Entscheidung auszurechnen und in einer Datenbank abzulegen. Bei Bedarf kann man in der Datenbank schnell nach einer konkreten Stellung suchen und bekommt die dazugehörige Lösung ausgegeben. Vier-Gewinnt [LVAvdH89], Go-Moku [All94] sowie Mühle [Gas95] sind auf diese Weise vollständig gelöst worden. Computer spielen diese Spiele perfekt.

Für Spiele, bei denen man nicht in der Lage ist, alle möglichen Situationen aufzuzählen und abzuspeichern, benutzt man oft Baumsuchverfahren, um gute Entscheidungen zu errechnen. Baumsuche ist ein sehr allgemeines Paradigma in der Informatik, das eine wichtige Rolle in vielen Anwendungen spielt. Sie bildet die Basis in Beweissystemen, Expertensystemen, Robotersteuerungen u.v.a.

Baumsuche kann auch beim Schachspiel, das weltweit wohl den größten Bekanntheitsgrad aller Spiele errungen hat, eingesetzt werden. Da das Schachspiel zwar eine kleine Welt aus wenigen, einfachen Regeln bildet, aber zum einen so komplex ist, daß es wohl niemals vollständig gelöst werden wird, und zum anderen oft herangezogen wird, um Aussagen über die Fähigkeiten strategischen und taktischen Denkens einer Person zu machen, haben wir dieses Spiel zur Beispielanwendung unserer Forschungen gemacht. Außerdem übt die Idee, eine Maschine zu konstruieren, die einen menschlichen Weltmeister im Schach schlägt, auf viele Menschen (u.a. auf den Verfasser dieser Arbeit) eine ungebrochene Faszination aus.

1.2 Computerschach

1.2.1 Die Anfänge

Für Levy beginnt die Geschichte des Computerschachs mit Charles Babbage (1792-1871). Der schrieb um 1840 einen Artikel für 'The Life of a Philosopher', aus dem hier aus Authentizitätsgründen ein kleiner Teil zitiert werden soll (entnommen aus Levy [LN91], Seite 25f):

'After much consideration I selected for my test the contrivance of a machine that should be able to play a game of purely intellectual skill successfully; such as tit-tat-to, drafts,

chess, etc.

I endeavoured to ascertain the opinions of persons in every class of life and of all ages, whether they thought it required human reason to play games of skill. The almost constant answer was in the affirmative. Some supported this view of the case by observing, that if it were otherwise, then an automaton could play such games. A few of those who had considerable acquaintance with mathematical science allowed the possibility of machinary being capable of such work; but they most stoutly denied the possibility of contriving such machinary on account of the myriads of combinations which even the simplest games included.

On the first part of my inquiry I soon arrived at a demonstration that every game of skill is susceptible of being played by an automaton. ...'

Er entwickelte im weiteren seine Ideen zur Vorausschau, die dem Minimaxprinzip sehr ähneln. Diese weiteren hier nicht aufgeführten Gedanken wirken aus heutiger Sicht zwar sehr schwerfällig, man sieht aber sehr deutlich an den Reaktionen, die er von den von ihm befragten Personen bekommt, wie revolutionär und futuristisch die Idee war, eine Maschine Schach spielen zu lassen.

Zu Beginn des 20. Jahrhunderts baute Torres y Quevedo eine mechanische Maschine, die in der Lage war, mit König und Turm den gegnerischen König mattzusetzen.

1.2.2 Modernes Computerschach

1945 beschrieb Zuse eine erste Formalisierung eines Schachspiels in seinem Werk 'Der Plankalkül' [Zus84].

1949 stellte Shannon [Sha50] ein Papier vor, dessen Brisanz darin liegt, daß die von ihm beschriebenen Grundlagen bis heute in jedem Computerschachprogramm vorhanden sind. Er formulierte eine sogenannte 'Typ-A-Strategie', bei der alle Varianten bis zu einer bestimmten Tiefe durchsucht, die Blätter durch heuristische Bewertungen evaluiert und die Ergebnisse nach dem Minimax-Prinzip zur Ausgangsstellung zurückgerechnet werden. Außerdem gab er eine 'Typ-B-Strategie' an, bei der einige Varianten tiefer untersucht werden als die übrigen. Er prägte auch den Begriff der Ruhesuche, in der nur Stellungen bewertet werden dürfen, die 'ruhig' sind. Beim Schachspiel z.B. sind ruhige Stellungen solche, in denen es keine Schlagzüge und keine einem Schach ausweichenden Züge auf dem Brett gibt.

In den späten 50er Jahren unseres Jahrhunderts wurde dann das erste Schachprogramm entwickelt. Es basierte auf den Vorschlägen von Shannon und Turing [Tur53].

Anfang der 60er Jahre waren Computer wegen ihrer Größe und der Anschaffungskosten ausschließlich beim Militär und bei militärnahen Einrichtungen zu finden. In dieser Zeit gab es keine nennenswerten Fortschritte im Computerschach. Gegen Ende der 60er Jahre

verfügten dann nahezu alle Universitäten und die großen Firmen über ein eigenes Rechenzentrum, das aus einem Großrechner und einer Vielzahl von Terminals bestand. In dieser Zeit entstanden die ersten Schachprogramme, wie wir sie heute kennen. Sie spielten Schach auf Anfängerniveau. 1967 erspielte sich ein Schachprogramm zum ersten Mal eine ELO-Zahl von 1640. Es war das Programm MACHACK [GEIC67] von Greenblatt, Eastlake und Crocker. Von da an nahm die Spielstärke von Schachprogrammen rasant zu, was hauptsächlich daran lag, daß die zugrunde liegende Hardware schneller wurde. Die Computerschachprogramme dieser Zeit wurden ausschließlich von Universitätsangehörigen geschrieben.

Ihrem akademischen Selbstverständnis folgend, veröffentlichten die Forscher und Entwickler von Schachprogrammen ihre Ergebnisse.

Zu Beginn der 80er Jahre eroberte der Personalcomputer die Welt. Die besten Schachprogramme wurden von da an von professionellen Schachprogrammierern geschrieben. Die Programme von Richard Lang dominierten über viele Jahre die Computerschachwelt im Bereich der PCs. Einem Vergleich mit den Programmen der Universitäten, die auf wesentlich leistungsfähigerer Hardware liefen, konnten sie allerdings nicht standhalten. Das schaffte aber 1992 Ed Schroeder, als er mit seinem Programm 'Chess Machine' die offene Weltmeisterschaft in Madrid vor allen Großrechnern gewann.

Im Gegensatz zu den akademischen Teams gibt es für die professionellen Computerschachprogrammierer gute Gründe, ihre Geheimnisse nicht preiszugeben. Im Gegenteil: Wissensvorsprung bedeutet Wettbewerbsvorteil. Seit 1990 gibt es nur noch sehr wenige Artikel und Arbeiten, die Bedeutung haben. Der Informationsaustausch verläuft über informelle Kanäle. Christian Donninger, der Programmierer des spielstarken Programms 'Nimzo', sieht die Zeit bis 1998 so [Don00]: 'Die etwas konnten, sagten nichts, und die etwas sagten, konnten nichts.'

Einige wenige ausgezeichnete Veröffentlichungen, die zu Durchbrüchen geführt haben, gab es allerdings doch: Da ist zum einen die Arbeit Donningers über die sogenannte Nullmovetechnik [Don93], die dafür sorgte, daß eine Reihe von Programmen Meisterniveau erreichen konnten. Zum anderen gibt es diejenigen Arbeiten, die sich mit der Parallelisierung von Spielbaumsuchverfahren auseinandersetzen, in erster Linie die Arbeiten von Rainer Feldmann [Fel93] und Peter Mysliwietz [Mys94].

1997 überraschte das IBM Programm Deep Blue mit der Sensation, Garry Kasparov, den zu der Zeit stärksten Schachspieler der Welt, mit 3,5 zu 2,5 zu schlagen. Allerdings geschahen die Arbeiten an Deep Blue ebenfalls im geheimen.

1999 beschrieb E.A. Heinz [Hei99] in seiner Dissertation den Status Quo im Computerschach und veröffentlichte mehrere heute gebräuchliche Methoden zur selektiven Suchsteuerung.

1.2.3 Weitreichende Bedeutung

Zeigt ein Schachcomputer Intelligenz?

Die Frage, ob Maschinen Intelligenz besitzen können, beschäftigt Forscher ebenso wie Science Fiction Autoren, seit es Computer gibt. Diese Frage ist natürlich noch ungelöst. Man muß aber immerhin anerkennen, daß Computer in der Lage sind, erstaunliche intellektuelle Leistungen zu erbringen. Dies zeigt sich besonders deutlich beim Schachspiel:

Schachspieler belächelten die ersten Schachprogramme, als diese vor 30 Jahren entwickelt wurden. Es galt als unmöglich, daß ein Computer 'gut' Schach spielen kann. Das Schachspiel galt sogar als Paradebeispiel für die überlegene menschliche Intelligenz. Heute hingegen ist kein Großmeister mehr vor einer Niederlage sicher, wenn er gegen eines der besten Schachprogramme antreten muß. Vorausschauen, planen, taktieren und Fehler konsequent ausnutzen können Schachprogramme mittlerweile genauso gut wie menschliche Schachgroßmeister. Der bisherige Höhepunkt des Computerschachs ist der berühmt gewordene Kampf des Schachprogramms Deep Blue gegen Garry Kasparov [JS97] [Sei97], bei dem sich der menschliche Weltmeister geschlagen geben mußte.

Die anfängliche Ablehnung des Computerschachs unter Schachspielern ist von konstruktiver Nutzung abgelöst worden.

Turnierschach

Entscheidende Bedeutung für das Turnierschach unter Menschen bekam der Schachcomputer schon Mitte der achtziger Jahre. Die besten käuflichen Schachprogramme hatten zu der Zeit das Niveau sehr guter Vereinsspieler erreicht. Mitte der neunziger Jahre spielten sie schon in etwa so stark wie Internationale Meister.

Nahezu jeder Schachspieler weiß seitdem die Objektivität und die nie endende Aufmerksamkeit seines Schachprogramms bei Analysen und Turniervorbereitungen zu schätzen.

Das Computerschach hat dem Schachspiel als solchem neue Impulse geben können. Jeder Schachspieler hat nämlich die Möglichkeit, zu Hause mit seinem eigenen Experten Stellungen und Partien zu analysieren. Es ist jetzt nicht mehr das Privileg einiger weniger, die das Glück haben, begabte Eltern oder Freunde zu haben, oder die zufällig in ein Förderprogramm gerutscht sind, gutes Schach spielen zu können. Mit ein wenig Übung kann jeder herausfinden, welche Züge gut und welche schlecht sind. Auf diese Weise ist Turnierschach auf breiter Front, von der Kreisliga bis zur Weltspitze, stärker geworden. Die Verbesserung vieler einzelner Entscheidungen mit Computerunterstützung (insbesondere in der Eröffnungsvorbereitung) hat zu einer Verbesserung des Gesamtsystems (d.h. zu höherwertigen Partien) geführt. Ein subjektiver Nebeneffekt ist, daß das Schachspiel wieder interessanter geworden ist. Mitte der 70er, Anfang der 80er Jahre galt Schach als

nahezu erforscht. Fast alle Spiele von Weltklassespielern endeten Remis. Der Computer hat erheblich dazu beigetragen, daß neue Wege entdeckt wurden, die das Spiel deutlich dynamischer erscheinen lassen, als bis dahin angenommen wurde.

Gesellschaft

Daß Manager von Firmen zur Entscheidungsunterstützung vorausschauende Software zur Verfügung bekommen, ist die Vision derjenigen, die sich interdisziplinär wissenschaftlich mit Strategieerforschung beschäftigen. Man darf dann darauf hoffen, daß wie im Schach auch die Verbesserung aller Einzelentscheidungen zu einer Verbesserung des Gesamtsystems führt. Diese Vision scheint in den letzten Jahren sehr viel näher gerückt zu sein, da sich offenbar auch in den Betriebswirtschaften der Gedanke durchsetzt, daß vorausschauendes Handeln einem rein rückwärts gerichteten Erfahrungshandeln überlegen sein könnte, besonders dann, wenn sich Rahmenbedingungen so rasant ändern, wie es im Moment der Fall ist. Man kann dieses z.B. daran erkennen, daß 1994 ein Nobelpreis an J.C. Harsanyi, J. F. Nash Jr. und R. Selten [Sel91] für die Verbindung von Spieltheorie und den Betriebswirtschaften verliehen wurde. Es sei allerdings angemerkt, daß jene Form der Spieltheorie nur wenig mit den hier vorgestellten, algorithmischen Ansätzen gemeinsam hat. Außerdem häufen sich Zeitungsartikel wie die im 'Schwerpunkt Unternehmenssteuerung' der Computerwoche [CW99], in denen über Softwaretools berichtet wird, die das Management von Firmen bei ihren Entscheidungen unterstützen sollen.

Computerschach ist aber auch noch von einem ganz anderen Standpunkt her interessant, nämlich aus der Sicht von Wissensmanagement. Dadurch, daß Partien gespielt werden, wird das Spiel von vorne nach hinten Stück für Stück analysiert und erforscht. Jeder, der Partieprotokolle beiträgt, trägt zur Erforschung des Spiels bei. Man kann also sagen, Tausende von Menschen in aller Welt erforschen zusammen das Schachspiel. Kommunikationsbasis ist dabei ein Schachdatenbank-System wie z.B. das der Firma ChessBase, die Partiedatenbanken, Analysewerkzeuge und sehr starke Schachprogramme vermarktet. Einen zweiten Pfeiler bildet das Internet, über das man über sogenannte Schachserver gegen Menschen in aller Welt Schach spielen kann. Unter

http://www.chess.co.uk/twic/twic.html

werden Partien von der Oberliga bis zur Weltspitze veröffentlicht. Schachserver ermöglichen rund um die Uhr das Spielen gegen Gegner aus aller Welt.

1.3 Algorithmen und Software

1.3.1 Minimax: Der Basisalgorithmus

Aus der klassischen Sichtweise der Informatik handelt es sich beim Schachspiel um ein Beispiel für ein Zweipersonen-Nullsummenspiel mit vollständiger Information und abwechselndem Zugrecht. Es gibt zwei Spieler, die gegensätzliche Interessen haben. Des einen Freud ist also des anderen Leid. Beiden ist das Spiel vollständig bekannt, d.h. beide kennen die aktuelle (Start-)Stellung, alle Züge zu jeder Stellung und die Bewertung von Endstellungen des Spiels.

Weist man nun den Spielern die Namen MAX und MIN zu (beim Schach Weiß und Schwarz), den Positionen, in denen MAX gewinnt (im Schach also den Positionen, in denen Weiß gerade Matt gesetzt hat) den Wert 1, den Positionen, in denen MIN gewinnt, den Wert -1 und den Remis-Stellungen den Wert 0 zu, kann man mit Hilfe des Minimax-Algorithmus zu jeder Stellung einen Wert bestimmen, der angibt, wer bei optimalem Spiel beider Seiten gewinnt.

Einer Endstellung x wird der Wert $f(x)$ zugewiesen:

$$f(x) = \begin{cases} 1 & \text{, falls } x \text{ für MAX gewonnen} \\ 0 & \text{, falls } x \text{ remis} \\ -1 & \text{, falls } x \text{ für MIN gewonnen} \end{cases}$$

Für eine beliebige Stellung y mit den Nachfolgestellungen $y_1 \dots y_{b(y)}$ läßt sich nun der Wert von y rekursiv berechnen, wenn man davon ausgeht, daß es keine unendlich langen Zugfolgen gibt.

$$F(y) = \begin{cases} \max\{F(y_1), \dots, F(y_{b(y)})\} & \text{, falls MAX am Zug ist} \\ \min\{F(y_1), \dots, F(y_{b(y)})\} & \text{, falls MIN am Zug ist} \end{cases}$$

Man nennt $F(y)$ den Minimaxwert der Stellung y. Um den in der Informatik üblichen Begriffen gerecht zu werden, bezeichnen wir im folgenden die Menge der Stellungen als die Menge V von *Knoten* und die Menge der Züge als die Menge $E \subset (V \times V)$ von *Kanten*. Wenn nun (V, E) ein gerichteter Baum ist und F eine Funktion, die Knoten auf Zahlen abbildet, wird $G = ((V, E), F)$ *Spielbaum* genannt. Knoten, die Endstellungen entsprechen werden als *Blätter*, der Knoten, der der Startstellung entspricht, wird als *Wurzel* ϵ *von* G bezeichnet.

Der Minimaxalgorithmus, der direkt aus der oben genannten rekursiven Definition von Werten hervorgeht, baut bei seiner Berechnung einen Spielbaum auf und wertet ihn aus.

1.3.2 Der $\alpha\beta$-Algorithmus

Wenn wir der Vereinfachung halber davon ausgehen, daß der entstehende Spielbaum b/t-uniform ist, d.h., jede Stellung genau b Nachfolger und jede Endstellung den Abstand t zur Startstellung hat, stellen wir fest, daß der Minimaxalgorithmus b^t viele Endstellungen besucht. Zu demselben Ergebnis, jedoch in vielen Fällen wesentlich schneller zur Lösung kommt der $\alpha\beta$-Algorithmus.

```
value alphabeta(node v, value α, value β)
1   generiere alle Nachfolger v_1, ..., v_b von v
2   if b = 0 return f(v) /* (Blattbewertung)*/
3   for i := 1 to b
4       if v ist ein MAX-Knoten {
5           α := max(α, alphabeta(v_i, α, β))
6           if α ≥ β return α
7           if i = b return α
8       } else {
9           β := min(β, alphabeta(v_i, α, β))
10          if α ≥ β return β
11          if i = b return β
12      }
```

Abb. 1.1: Grundform des $\alpha\beta$-Algorithmus

1975 haben Knuth und Moore [KM75] gezeigt, daß für jeden Knoten v des Spielbaums folgende Ungleichungen gelten:

$$\begin{array}{ll} \text{alphabeta}(v,\alpha,\beta) \leq \alpha & \text{falls } F(v) \leq \alpha \\ \text{alphabeta}(v,\alpha,\beta) = F(v) & \text{falls } \alpha < F(v) < \beta \\ \text{alphabeta}(v,\alpha,\beta) \geq \beta & \text{falls } F(v) \geq \beta \end{array}$$

Deshalb berechnet der Aufruf von alphabeta$(v, -\infty, \infty)$ den Wert minimax(v). Bei optimaler Sortierung der generierten Knoten inspiziert dieser Algorithmus lediglich $b^{\lfloor t/2 \rfloor} + b^{\lceil t/2 \rceil} - 1$ Blätter.

In seinem Buch [Rei89] präsentiert Reinefeld eine Reihe von Varianten des $\alpha\beta$-Algorithmus, die noch einfachere Programmierung erlauben und in den getesteten Anwendungen die Anzahl der gesuchten Knoten verringern.

Zwei zusätzliche Heuristiken haben dem $\alpha\beta$-Algorithmus zu seinem Siegeszug beim Schachspiel verholfen: die sogenannten Transpositionstabellen, das sind Hashtabellen, die es ermöglichen, die Zugsortierung enorm zu verbessern, und die sogenannte Nullmoveheuristik, die es erlaubt, mit zu vernachlässigbarem Aufwand große Teilbäume während der $\alpha\beta$-Suche abzuschneiden.

Eine Übersicht über die modernsten und erfolgreichsten Varianten der $\alpha\beta$-Suche findet man in der Doktorarbeit von E.A. Heinz [Hei99].

In praktischen Anwendungen, wie z.B. im Schachspiel, ist es aufgrund der Größe der entstehenden Suchbäume allerdings nur selten möglich, das Spiel auf die eben vorgestellte Weise erschöpfend zu untersuchen. Deshalb behilft man sich mit einem Trick: Man untersucht nicht den vollständigen Schach-Spielbaum, sondern nur einen kleinen Teil davon. Statt der tatsächlichen Stellungswerte (die man nicht zur Verfügung hat) werden an den Blättern dieses Baums Schätzwerte eingesetzt, die dann nach dem oben beschriebenen Minimax-Prinzip ausgewertet werden. Jahrelange Beobachtungen zeigen, daß in vielen Spielen größere Suchbäume zu besserer Entscheidungsqualität führen, obwohl die Qualität der Schätzwerte an allen Knoten gleich ist.

1.4 Näherungslösungen

Für einige Spiele hat man zeigen können, daß sie PSPACE-vollständig sind. Die Konsequenz ist, daß man (zumindest solange die Polynomzeithierarchie bestehen bleibt) nichts Besseres tun kann, als einen Spielbaum auszuwerten, um eine perfekte Entscheidung treffen zu können. Da die entstehenden Spielbäume im allgemeinen viel zu groß sind, um sie vollständig zu untersuchen, und da man die echten Werte eines Spiels normalerweise nicht kennt, ist man gezwungen, seine Entscheidungen auf unscharfes und unsicheres Wissen zu stützen.

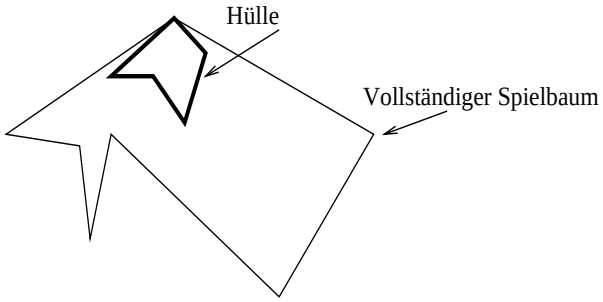


Abb. 1.2: Nur die Hülle wird von einem Suchalgorithmus abgesucht.

Eine Approximation einer Entscheidung in einem Zweipersonen-Nullsummenspiel wird

in der Praxis folgendermaßen errechnet:

Zunächst einmal wird ein Teilbaum, der die gleiche Startstellung besitzt wie der eigentliche Spielbaum, für eine Untersuchung ausgewählt. Wir bezeichnen diesen Teilbaum im folgenden immer als Hülle. Ein geeignetes Spielbaumsuchverfahren versieht die Blätter der Hülle mit heuristischen Werten und rechnet diese Werte nach dem Minimax-Prinzip zur Wurzel hoch. In fast allen Fällen wird eine Variante des $\alpha\beta$-Algorithmus eingesetzt, da dieser zusammen mit einer Reihe von Sortierheuristiken [Sch89b], [Fel93] ein sehr gutes Zeitverhalten besitzt.

Eine Näherungslösung durch Hüllen, die an jedem inneren Knoten alle Nachfolger enthalten, die auch im Originalspielbaum enthalten sind und deren Blätter alle eine Entfernung t zur Wurzel haben, liefert in vielen Fällen gute Resultate. Allerdings hat man mittlerweile Heuristiken gefunden, um eine Hülle individueller zu gestalten und damit die durchschnittliche Qualität von Entscheidungen deutlich anzuheben. Einige dieser Techniken sind anwendungsunabhängig wie z.B. die Singular Extensions [Ana91], die Nullmoves [Bea89] [Don93] oder die Fail High Reduktionen [Fel96]. Viele andere sind anwendungsabhängig. Man ist sich in Computerschachkreisen einig, daß die Form der Hülle ganz entscheidende Bedeutung für die Qualität der Suche hat [Don95].

Man kann zwei Klassen von Spielbaum-Suchalgorithmen unterscheiden. Auf der einen Seite gibt es diejenigen, die einen Minimaxwert einer zuvor fest gewählten Hülle errechnen: Der $\alpha\beta$-Algorithmus, der SCOUT-Algorithmus [Pea84] [MRS87] oder SSS* [Sto79] sind in den letzten 30 Jahren erschöpfend untersucht worden. Eine Arbeit von Plaat [Pla93] über den MTD(f) Algorithmus zeigt, daß man den SSS* Algorithmus durch eine Folge von $\alpha\beta$-Nullfenstersuchen simulieren kann.

Auf der anderen Seite stehen sogenannte *iterative Suchheuristiken* [Riv87], die einen Spielbaum in jeder Zeiteinheit einen Schritt 'wachsen' lassen. In jedem Schritt wird ein Blatt ausgewählt (Auswahlphase), und die Nachfolger dieses Blattes werden dem Spielbaum hinzugefügt (Expansion). Dann werden die neuen Blätter bewertet, und der neue heuristische Minimaxwert wird von den Blättern zur Wurzel aktualisiert (Aktualisierungsphase).

Die auf diese Weise gewachsenen Spielbäume brauchen weder tiefenuniform zu sein, noch ist es notwendig, die Hülle vor Beendigung der Spielbaumsuche festzulegen. Beispiele für solch iterative Suchheuristiken sind Berliners B* Algorithmus [Ber79], Palays wahrscheinlichkeitsbasierte Methode [Pal85] und Conspiracy Number Search. Letztere wurde von D. McAllester [McA88] vorgeschlagen. J. Schaeffer [Sch90] griff die Idee auf und implementierte einen Algorithmus, der sich als sehr gut im Bereich taktischer Suchen beim Schach erwiesen hat. U. Lorenz und andere [LRFM95] haben bereits 1995 Ide-

en präsentiert, die es ermöglichen sollten, Conspiracy Number Suchen effizienter durchzuführen.

Der Ausgangspunkt für Conspiracy Number Search (CNS) ist die Beobachtung, daß der $\alpha\beta$-Algorithmus in gewissem Sinn Entscheidungen mit niedriger Sicherheit liefert:
Im allgemeinen kann nämlich die Änderung eines einzelnen Blattwertes (z.B. durch einen Fehler der heuristischen Bewertungsfunktion) zu einer Änderung der Entscheidung führen, die an der Wurzel gefällt werden muß. Der $\alpha\beta$-Algorithmus fällt also Entscheidungen mit Sicherheit (d.h. mit Conspiracy Number) eins.
Das Ziel von CNS ist es, die vorhandenen Ressourcen so zu verteilen, daß garantiert ist, daß eine Entscheidung mit einer Conspiracy Number $c > 1$ gefällt wird. Das bedeutet dann, daß eine Entscheidung an der Wurzel stabil ist, gegenüber einer willkürlichen Veränderung von bis zu $c - 1$ Blattwerten. Schaeffers Algorithmus verwaltet die nötigen Informationen mit Hilfe von Vektoren, die er an allen Knoten des Spielbaums speichert.
Ein Conspiracy Number Vektor eines Knotens v informiert darüber, wieviele Knoten unterhalb von v ihren Wert zu einer Zahl x verändern müssen, damit sich der Minimaxwert von v auf den Wert x ändert. Weil alle gesammelten Informationen jederzeit verfügbar sein müssen, wird der Speicherbedarf des Verfahrens durch die Anzahl der untersuchten Knoten und vor allem durch die Granularität der Bewertungsfunktion bestimmt.
Der enorme Speicherbedarf ist auch einer der Gründe, weshalb sich der Einsatz von CNS bisher auf taktische Suchen beschränkt hat. Schaeffer [Sch90] hat versucht, mit Hilfe einer grob-granularen Bewertungsfunktion Entscheidungen zu finden, die deutlich besser erscheinen als die Alternativen. Für taktische Stellungen hat sich CNS den uniform gestalteten Suchen als überlegen erwiesen.

Allgemeine Eingaben erfüllen allerdings nicht die Eigenschaft, daß klar überlegene Entscheidungen vorhanden sind. Die Spielbaumsuche auf Instanzen, bei denen eine Entscheidung zu finden ist, die nur marginal besser ist als ihre Alternativen, bezeichnet man auch als positionelle oder strategische Suche. Zu diesem Zweck benötigt man eine feingranulare Bewertungsfunktion und damit auch sehr viel Speicher. Es ist enttäuschend zu sehen, wie die konventionelle CNS ernste Probleme mit der Terminierung bekommt, wenn man eine fein-granulare Bewertung einsetzt. Manchmal untersucht CNS schon riesige Teilbäume, um eine Entscheidung auch nur mit niedriger Sicherheit zu finden. Als Konsequenz haben diese Nachteile dazu geführt, daß die CNS nicht erfolgreich für allgemeine Eingabeinstanzen implementiert wurde.

1.5 Conspiracy Number Search

1.5.1 Conspiracy Numbers (dt. Verschwörungszahlen)

In jedem Spielbaum, dessen Wert so zustande kommt, daß man seine Blätter mit heuristischen Werten belegt, die man nach dem Minimax-Prinzip zur Wurzel hochrechnet, hängt der Wert der Wurzel von den heuristischen Blattwerten ab. Wenn ein Spielbaum nun so aufgebaut wird, daß bestehende Blätter eines Spielbaums zu inneren Knoten gemacht werden können, indem alle Nachfolger dieses Blattes erzeugt und heuristisch bewertet werden, kann es passieren, daß sich nach solch einem Erweiterungsschritt der Wert der Wurzel ebenfalls ändert. Der Effekt, den Blattwertveränderungen auf einen Wurzelwert haben können, ist die Rechtfertigung der Conspiracy Numbers, die von McAllester [McA88] 1988 erstmals vorgestellt wurden. Eines der Hauptziele ist die Bekämpfung einiger Schwächen des $\alpha\beta$-Algorithmus: In der Grundversion schneidet der $\alpha\beta$-Algorithmus seinen Suchbaum in einer festen Tiefe ab, unabhängig von der Wichtigkeit bestimmter Varianten. Im Extremfall kann es passieren, daß das Ergebnis des Algorithmus an der Wurzel des Spielbaums von einer einzigen statischen Bewertung eines Blattes abhängt.

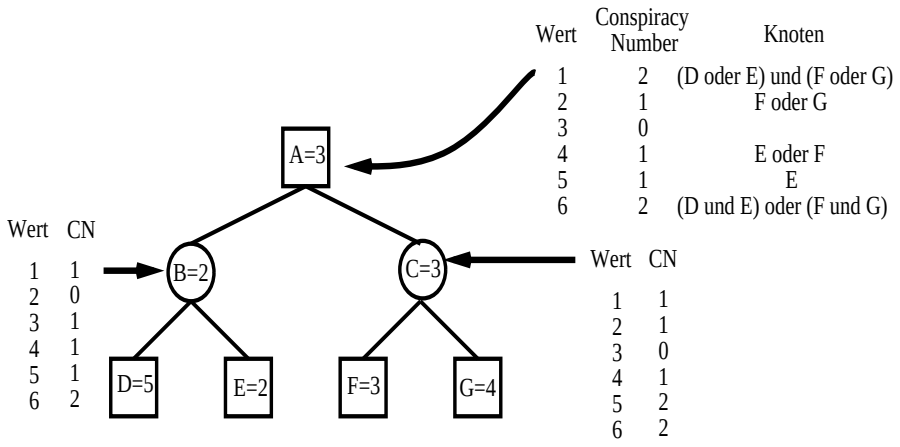


Abb. 1.3: Conspiracy Numbers

Die originale Definition einer Conspiracy Number (dt. Verschwörungszahl) ist wie folgt:

Definition 1.5-1 **(Conspiracy Number)**
Sei $T = (V, E)$ ein gerichteter Baum und h eine heuristische Bewertungsfunktion, die Knoten auf Zahlen abbildet. Die Conspiracy Number der Wurzel eines Spielbaums $G = ((V, E), h)$ für einen Wert x ist definiert als die minimale Anzahl der Blätter von G, die

ihren Wert zu x ändern müssen, um den Minimaxwert der Wurzel von G zu x zu ändern. □

Eine Conspiracy Number n besagt, daß sich mindestens n Blattwerte ändern (d.h. 'verschwören') müssen, um den Wert der Wurzel zu x zu verändern. Betrachten wir zur Verdeutlichung das Beispiel in Abb. 1.3 [Sch90]. Die Anzahl der Nachfolger jedes inneren Knotens sei dabei 2, die möglichen Werte seien 1 bis 6. Innerhalb der Knoten sind der jeweilige Name und der Minimaxwert eingezeichnet. Neben den inneren Knoten ist jeweils eine Tabelle zu sehen, die zeigt, wieviele Blätter ihren Wert mindestens ändern müssen, um den Wert der Wurzel zu 1,2,3,4,5 oder 6 zu ändern. So braucht sich z.B nur der Wert des Blattes E auf 5 zu ändern, und schon wird auch der Wert der Wurzel 5.
Es gibt eine einfache Möglichkeit, die Conspiracy Numbers rekursiv zu berechnen [Sch90] [vdM90].

- Die Conspiracy Number eines Blattes ist 0 für den Wert des Blattes. Für alle anderen Werte ist die Conspiracy Number 1.
- Wenn wir den Wert x eines MAX-Knotens (MIN-Knotens) v auf $x' < x$ senken ($x' > x$ erhöhen) wollen, müssen alle Nachfolger von v, die einen Wert größer (kleiner) als x' haben, ihren Wert absenken (erhöhen). Wenn $v_1 \ldots v_c$ diese Nachfolger sind, wobei $cn(v_1, x') \ldots cn(v_c, x')$ gerade die Conspiracy Numbers der jeweiligen Knoten für den Wert x' sind, ist die Conspiracy Number des Knotens v für den Wert x' offensichtlich die Summe der $cn(v_1, x') \ldots cn(v_c, x')$.
- Wenn wir den Wert x eines MAX-Knotens (MIN-Knotens) v auf $x' > x$ erhöhen ($x' < x$ vermindern) wollen, reicht es, wenn ein Nachfolger von v den Wert x' annimmt. Seien $cn(v_1, x') \ldots cn(v_c, x')$ die Conspiracy Numbers der jeweiligen Knoten für den Wert x'. Die Conspiracy Number von v für x' ist das Minimum der $cn(v_1, x') \ldots cn(v_c, x')$.

Abbildung 1.4 zeigt den Sachverhalt formal. Minimax(v) bezeichnet den Minimaxwert des Teilbaums, dessen Wurzel v ist, $cn(v, x)$ bezeichnet die Conspiracy Number (Verschwörungszahl) von v für den Wert x.

1.5.2 Conspiracy Number Search (CNS)

Conspiracy Numbers sind ein Maß für die Sicherheit, daß der Wert der Wurzel richtig eingeschätzt wird. Das Ziel einer Suche muß es deshalb sein, einen Suchbaum so aus dem Gesamtspielbaum herauszuschneiden, daß der Wurzelwert mit einer vorgegebenen Conspiracy Number sicher ist.

if minimax(v) = x
    cn(v,x) := 0;
else if v ist ein Blatt {
    cn(v,x) := 1;
} **else** {
    if v ist ein MAX-Knoten und $x <$ minimax(v)
        cn(v,x) := $\sum_{v_j \in \Gamma(v), x < \text{minimax}(v_j)}$ cn(v_j,x);
        /* $\Gamma(v)$ bezeichne die Menge aller Nachfolger von v */
        /* v_j sei der j-te Nachfolger */
    else if v ist ein MIN-Knoten and $x >$ minimax(v)
        cn(v,x) := $\sum_{v_j \in \Gamma(v), x > \text{minimax}(v_j)}$ cn(v_j,x);
    else if (v ist ein MAX-Knoten und $x >$ minimax(v))
            oder (v ist ein MIN-Knoten und $x <$ minimax(v))
        cn(v,x) := $\min_{v_j \in \Gamma(v)}$ cn(v_j,x);
}

Abb. 1.4: Rekursive Berechnung der Conspiracy Numbers cn(v,x)

In den von Schaeffer und van der Meulen [vdM90] vorgestellten Algorithmen, denen eine Sicherheitsschranke c vorgegeben wird, wird folgendermaßen vorgegangen: Wenn der Wert einer Wurzel so gesichert ist, daß sich c Blätter verändern müssen, um den Wurzelwert überhaupt zu beeinflussen, gilt der Wurzelwert als sicher, und der Algorithmus terminiert. Solange es einen Wert gibt, dessen zugehörige Conspiracy Number kleiner als c ist, werden die folgenden drei Phasen durchlaufen (vgl. Abbildung 1.5):

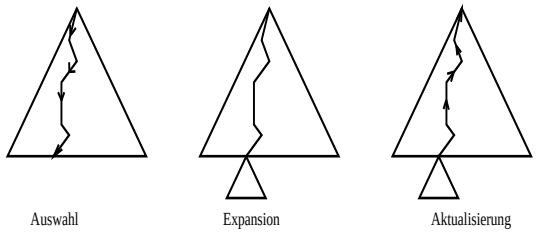


Abb. 1.5: Auswahl, Expansion und Aktualisierung

Zunächst läuft der CNS-Algorithmus von der Wurzel zu einem Blatt, das zu der Menge von Blättern gehört, die den Wurzelwert auf x bringen können. Diese erste Phase wird *Auswahlphase* genannt. Der Algorithmus findet ein solches Blatt mit Hilfe der an den Knoten gespeicherten Conspiracy Number Vektoren. Es ist möglich, aufgrund rein lokaler

Entscheidungen einen Pfad $P = (v_1 = \epsilon, \ldots, v_n)$ von der Spielbaumwurzel zu einem geeigneten Blatt v_n zu finden. Wie der Auswahlprozeß abläuft, betrachten wir am besten an dem Beispiel in Abb. 1.3.

Sei eine Sicherheitsschranke $c = 2$ vorgegeben. An einem Knoten v werden Werte, deren Conspiracy Numbers kleiner als c sind, als *plausible* Werte bezeichnet. Der Wert, für den die Conspiracy Number an v gleich Null ist, wird als der *plausibelste* Wert angesehen. Der vorliegende Spielbaum soll so erweitert werden, daß die Conspiracy Number des am weitesten vom Wurzelwert entfernten plausiblen Wertes (V_p) erhöht wird. Dazu wird dasjenige Blatt des Spielbaums erweitert (d.h., es werden alle Nachfolger davon generiert und bewertet), das am ehesten in der Lage zu sein scheint, den Wert der Wurzel auf V_p zu ziehen. (Falls es mehrere gleichwertige Kandidaten gibt, wird der linkeste der Kandidaten erweitert.)

In unserem Beispiel beginnt die Auswahlphase bei der Wurzel A. Die Werte 1 und 6 der Wurzel in Abb. 1.3 werden als nicht in Frage kommend betrachtet, weil ihre Conspiracy Numbers schon größer oder gleich c sind. Der kleinste plausible Wert (V_{min}) ist 2, der größte plausible Wert (V_{max}) ist 5, und der plausibelste Wert (V_{Wurzel}) ist 3. Es gibt mehr plausible Werte oberhalb von V_{Wurzel}, als darunter ($V_{max} - V_{Wurzel} > V_{Wurzel} - V_{min}$). Deshalb versucht der Algorithmus, den Bereich der plausiblen Werte oberhalb von V_{Wurzel} zu verkleinern. Er erweitert den Knoten, der am ehesten in der Lage zu sein scheint, den Wert der Wurzel zu 5 zu ändern. Aus Sicht der Wurzel hat der Knoten B nur eine Conspiracy Number von 1 für den Wert 5, der Knoten C eine Conspiracy Number von 2. Es hat den Anschein, als müsse man unter B weniger Arbeit investieren, um den Wert der Wurzel, wenn überhaupt, auf 5 zu bringen. Der Auswahlprozeß verzweigt deshalb zum Knoten B. Danach wird nach E verzweigt, da eine Änderung des Wertes von E auf 5 den Wert der Wurzel zu 5 werden ließe, eine Änderung (in diesem Fall eine Beibehaltung) des Wertes von D auf 5 keine Auswirkung hätte.

Die *Erweiterungsphase* erzeugt und bewertet alle Nachfolger des Blattes, das in der Auswahlphase ausgewählt wurde.

In der dritten Phase werden die neuen Erkenntnisse der Erweiterungsphase in den Suchbaum eingebunden. Man spricht von der *Aktualisierungsphase*. Die Werte der neuen Blätter werden zur Wurzel hin nach dem Minimax-Prinzip ausgewertet.

Die Algorithmen von Schaeffer und van der Meulen suchen sehr selektiv schmale Varianten ab. Leider erweitern sie den Suchbaum auch ab und an in unnötige Tiefen, um etwas zu beweisen, was nicht zu beweisen ist. Das Konvergenzverhalten des Verfahrens ist manchmal sehr langsam, bzw. wenn man unendlich große Instanzen zuläßt, kann es passieren, daß die Verfahren ihre Berechnungen niemals beenden. Beide Implementationen der Conspiracy Number Suche zeigen gutes Verhalten auf taktischen Schachstellungen, zeigen aber auch Schwächen:

- Die Erweiterung des Suchbaums ist sehr aufwendig, weil keine vorgegebenen Schranken wie z.B. beim $\alpha\beta$-Verfahren vorhanden sind.
- An jedem Knoten müssen Conspiracy Number Vektoren gespeichert werden. Der Speicherverbrauch hängt deshalb stark von der Granularität der benutzten Bewertungsfunktion ab.
- Strategisches Positionsschach ist kaum möglich, weil das Konvergenzverhalten bei Benutzung einer fein-granularen Bewertungsfunktion sehr schlecht ist.
- Dadurch, daß in jedem Metaschritt (Auswahl + Expansion + Aktualisierung) der Algorithmen der jeweilige Suchbaum an einem Blatt erweitert wird, sind beide Implementationen inhärent sequentiell.
- Wie bei allen Bestensuchverfahren muß der Suchbaum vollständig im Speicher gehalten werden. Die Speicheranforderungen sind somit linear in der Suchzeit.

1.5.3 Beobachtungen

Beobachtung 1.5-1
Sei $P = (\epsilon = v_1, \ldots, v_n)$ der Pfad der t-ten Auswahlphase. Dann gibt es häufig ein $v_j, 1 \leq j \leq n$, dessen Conspiracy Number Vektor durch die folgende Spielbaumerweiterung gar nicht beeinflußt wird. Deshalb sind $v_1 \ldots v_j$ auch Teil des Pfades der $(t+1)$-sten Auswahlphase. □

Beobachtung 1.5-2
Eine MAX(/MIN)-*Strategie* in einem Spielbaum G ist ein Teilbaum von G (mit derselben Wurzel wie G), der an jedem MAX(/MIN)-Knoten genau einen und an jedem MIN(/MAX)-Knoten alle Nachfolger, die es entsprechend G gibt, enthält. Eine MAX-(/MIN)-Strategie liefert eine untere(/obere) Schranke für den Minimaxwert von G.

Der Begriff der Conspiracy Numbers läßt sich auch mit Hilfe des Strategiebegriffs ausdrücken. Wenn nämlich mindestens n Blätter ihren Wert zu x ändern müssen, um den Wurzelwert zu x zu ändern, gibt es n blattdisjunkte Strategien, die zeigen, daß der Wert der Wurzel nicht x ist. Gibt es umgekehrt n blattdisjunkte Strategien, die zeigen, daß der Wert größer (bzw. kleiner) einer bestimmten Schranke x' ist, müssen sich mindestens n Blattwerte ändern, damit der Minimaxwert der Wurzel die gegebene Schranke verletzt (s. Abschnitt 2.1.1). Deswegen werden wir den Begriff der Conspiracy Numbers im nächsten Kapitel für Schranken definieren. □

Beobachtung 1.5-3
Die gerade gemachte Beobachtung impliziert die folgende: Wenn ein konventioneller CNS-Algorithmus einen Wurzelwert x 'einschnürt' und nacheinander 'beweist', daß der Wert nicht $x_c > x_{c-1} > \cdots > x$ ist, sucht er nacheinander Strategien, die zeigen, daß der Wert der Wurzel $\leq x_c, \leq x_{c-1}, \ldots, \leq x$ ist. Eine Strategie, die beweist, daß der Wurzelwert $\leq x$ ist, beweist aber schon, daß der Wert auch $\leq x_c$ ist. □

Beobachtung 1.5-4
Es genügt, an der Wurzel eines Spielbaums einen besten Zug zu finden. Der absolute Wert der dazugehörigen Wurzel ist dann nicht wichtig. Es genügt zu wissen, daß der Minimaxwert des einen Wurzelnachfolgers besser ist als die Minimaxwerte der anderen Wurzelnachfolger. □

1.6 Ergebnisse dieser Arbeit

In dieser Arbeit werden wir ein neuartiges Spielbaumsuchverfahren vorstellen, das wir auch erfolgreich parallelisiert haben.

Es handelt sich um einen von uns so genannten Controlled Conspiracy Number Search Algorithmus, der eine wesentlich zielgerichtetere Suche darstellt als die herkömmliche Conspiracy Number Search. Er löst die im vorigen Abschnitt skizzierten Probleme soweit, daß ein sehr erfolgreicher Einsatz in einem Schachprogramm möglich ist. Der Suchalgorithmus wird von sogenannten Targets kontrolliert, die während der Suche Anforderungen an Knoten definieren.

Das neue Spielbaumsuchverfahren wird im Weltklasse-Schachprogramm P.ConNerS verwendet, das u.a. als erstes Programm der Welt ein offizielles Schachgroßmeisterturnier unter regulären Turnierbedingungen gewinnen konnte.
Beim 10. Lippstädter Großmeisterturnier 2000 lief P.ConNerS auf einem System mit 160 Pentium II 450MHz Prozessoren. Damit ist der in dieser Arbeit vorgestellte parallele Algorithmus der erste, der in einem erfolgreichen Schachprogramm auf einem Workstationcluster dieser Größe effizientes Verhalten zeigt.

Eine theoretische Analyse von Blattfehlerbewertungen in Spielbäumen verhilft zu der Einsicht, daß die Spielbaumstruktur entscheidenden Einfluß auf die Fortpflanzung solcher Fehler besitzt. Sie motiviert noch einmal die Conspiracy Number Idee.

1.7 Aufbau dieser Arbeit

Im zweiten Kapitel zeigen wir zunächst (Abschnitt 2.1.1), daß man Conspiracy Numbers auch über den Begriff von 'blattdisjunkten Strategien', die den Wert der Spielbaumwurzel belegen, definieren kann.

In den folgenden Abschnitten, beginnend bei Abschnitt 2.1.2, beschreiben wir den Cc2s-Algorithmus, der folgendes Sicherheitskonzept umsetzt: Um an der Wurzel eines Spielbaums zu einer Entscheidung zu kommen, müssen wir eine untere Schranke für den Wert des besten Wurzelnachfolgers bestimmen und obere Schranken für die Werte der anderen Nachfolger. Das Ziel ist es, eine Hülle nach dem Minimax-Prinzip auszuwerten, deren Blätter eine Entfernung von mindestens t zur Wurzel haben und die gleichzeitig mindestens zwei blattdisjunkte Strategien für jede, wie gerade beschriebene, zu berechnende Schranke enthält. (Man vergleiche diese Zielsetzung mit dem dritten Kapitel über Fehlerfortpflanzung in Spielbäumen.)

Eine Erweiterung des Verfahrens mit fester Conspiracy Number 2 zu einem allgemeinen Controlled CNS-Algorithmus ist leicht möglich.

Es folgen im Abschnitt 2.2 einige hergeleitete Eigenschaften des Verfahrens, und wir vergleichen es soweit möglich mit dem $\alpha\beta$-Algorithmus und mit konventionellen inkrementellen Suchalgorithmen. Dabei machen wir die folgenden Beobachtungen:

a) Wenn der Cc2s-Algorithmus terminiert ist, basiert das Ergebnis auf einer Hülle H mit den gewünschten Eigenschaften, und dieses Ergebnis basiert auf Minimaxwerten der Hülle H . D.h. bei vorgegebener Hülle H käme jede Variante eines Minimax-Algorithmus zu demselben Ergebnis.

b) Wenn wir uns auf vordefinierte und endliche Hüllen beschränken (was z.B. für Analysen des $\alpha\beta$-Algorithmus immer getan wurde), terminiert unser Algorithmus in endlicher Zeit.

c) Wenn wir wie in b) nur vordefinierte und endliche Hüllen betrachten und somit nicht zwei blattdisjunkte Beweisstrategien als Sicherheit fordern, untersucht unser Algorithmus im besten Fall die minimal mögliche Anzahl Blätter.

Danach richten wir unsere Aufmerksamkeit auf das Verhalten des Verfahrens in einer Anwendung, und zwar in unserem Schachprogramm ConNerS. Zunächst werden zu diesem Zweck einige praxisrelevante Zusatzheuristiken zur Beschleunigung der Suche kurz vorgestellt (Abschnitt 2.3). Abschnitt 2.4 zeigt die experimentellen Ergebnisse in der Anwendung Schach.

Kapitel 3 beschäftigt sich mit der Parallelisierung des neuen Verfahrens. Abschnitt 3.1 beschreibt die verwendeten Hardwareplattformen. Abschnitt 3.2 gibt einen Überblick über die wichtigsten Veröffentlichungen zum Thema parallele bzw. verteilte Spielbaumsuche.

Der Rest von Kapitel 3 beschreibt die Parallelisierung des Cc2s-Algorithmus (Abschnitt 3.3) und dazugehörige experimentelle Ergebnisse (Abschnitt 3.5).

Kapitel 4 beschreibt eine Analyse von Fehlerfortpflanzungen in Spielbäumen mit nicht-perfekter Bewertungsfunktion. Wir werden dort zeigen, daß der Begriff der 'blattdisjunkten den Wurzelwert belegenden Strategien' sowohl unter Worstcasebetrachtungen als auch unter Durchschnittsbetrachtungen von entscheidender Bedeutung ist. Die Analyse motiviert noch einmal die Verwendung von Conspiracy Numbers.

1.8 Definitionen und Begriffe

1.8.1 Allgemeine Definitionen

Definition 1.8-2 **(Spielbaum)**
$G = (T, h)$ ist ein *Spielbaum*, wenn $T = (V, K)$ ein gerichteter Baum ist und $h : V \to \mathbb{Z}$ eine Funktion von der Menge der Knoten in die Menge der ganzen Zahlen. $L(G)$ bezeichnet die Menge der Blätter von T. $\Gamma(v)$ ist die Menge der Nachfolger des Knotens v. Knoten des Baums T bezeichnen wir auch als Knoten des Spielbaums G.

Bemerkung: Wir identifizieren Knoten eines Spielbaums G mit Stellungen des vorliegenden Spiels und Kanten von T mit Zügen von einer Stellung zur nächsten. Außerdem gibt es zwei *Spieler*: MIN und MAX, die abwechselnd ziehen (d.h. eine Kante wählen). Wenn wir nicht explizit etwas anderes voraussetzen, gehen wir davon aus, daß die Wurzel ein MAX-Knoten ist. D.h., MAX zieht an den Knoten in den geraden Ebenen von T, MIN an den Knoten in den ungeraden Ebenen. □

Definition 1.8-3 **(Hülle)**
Es sei G ein Spielbaum mit Wurzel ϵ. Ein Teilbaum H von G heißt *Hülle* von G, wenn gilt:

- $\epsilon \in H$
- $\forall v \in H : \Gamma_H(v) = \Gamma_G(v)$ oder $\Gamma_H(v) = \emptyset$.

□

Bemerkung: Sei $\mathcal{A}$ ein Suchalgorithmus für Spielbäume. Wir unterscheiden nicht nur zwischen Spielbaum und Hülle, sondern zusätzlich noch zwischen Hülle und (aktuellem) *Suchbaum*. Ein Suchbaum ist ein Teilbaum einer Hülle. So könnte zum Beispiel

ein in Breite und Tiefe uniformer Teilbaum eines Gesamtspielbaums als Hülle dienen. Der Minimax-Algorithmus und der $\alpha\beta$-Algorithmus kämen bei der Auswertung dieser Hülle zwar zu dem gleichen Ergebnis, würden aber verschiedene Suchbäume erzeugen. Sei v ein Knoten und sei $\Gamma(v)$ die Menge der Nachfolger von v, eine Hülle H betreffend. $\Gamma'(v)$ bezeichnet die Menge derjenigen Nachfolger von v, die explizit vom Algorithmus $\mathcal{A}$ zu einem Zeitpunkt t untersucht wurden. Streng genommen müßten wir also $\Gamma'(v)$ als $\Gamma'_{\mathcal{A},t}(v)$ schreiben. Aus Gründen der Übersichtlichkeit verzichten wir, falls es zu keinen Verwechslungen kommen kann, auf die Mehrfachindizierung.

Definition 1.8-4 **(Minimaxwerte)**
Sei $G = (T, h)$ ein Spielbaum und $v \in V$ ein Knoten von T. Die Funktion minimax : $V \to \mathbb{Z}$ ist rekursiv definiert durch

$$\text{minimax}(v) := \begin{cases} h(v) \quad \text{falls } v \in L(G) \\ \max\{\text{minimax}(v') \mid (v, v') \in E\} \\ \quad \text{falls } v \notin L(G) \text{ und MAX ist am Zug} \\ \min\{\text{minimax}(v') \mid (v, v') \in E\} \\ \quad \text{falls } v \notin L(G) \text{ und MIN ist am Zug} \end{cases}$$

Wir nennen minimax(v) den *Minimaxwert* von v. Der Minimaxwert der Wurzel von G wird auch als minimax(G) bezeichnet. □

Definition 1.8-5 **(Strategie)**
Sei $G = (T, h)$ ein Spielbaum (oder eine Hülle), $s \in \{\text{MIN}, \text{MAX}\}$ mit der Wurzel $\epsilon \in V$. Für einen beliebigen Knoten $u \in V$ sei $\Gamma(u)$ die Menge der Nachfolger von u im Spielbaum G. Eine *Strategie für einen Spieler* s ist ein Teilbaum $S_s = (V_s, E_s)$ des Spielbaums G, für den gilt:

- $\epsilon \in V_s$
- Falls $u \in V_s$ ein innerer Knoten von T ist, bei dem s am Zug ist, gibt es genau einen Nachfolger $u' \in \Gamma(u)$ von u mit $u' \in V_s$ und $(u, u') \in E_s$.
- Falls $u \in V_s$ ein innerer Knoten von T ist, bei dem der Gegner $\overline{s}$ von s ($\overline{s} \in \{\text{MAX,MIN}\} - \{s\}$) am Zug ist, gilt $\Gamma(u) \subset V_s$ und für alle $u' \in \Gamma(u)$ ist $(u, u') \in E_s$.

Eine MAX-Strategie (vgl. auch Abb. 1.6) ist damit ein Teilbaum mit Wurzel ϵ von G, der an jedem MAX-Knoten genau einen und an jedem MIN-Knoten alle Nachfolger von G enthält. Für MIN-Strategien ist es genau anders herum. Knoten, die innerhalb einer Strategie genau einen Nachfolger besitzen, nennen wir CUT-Knoten. Die Knoten, an denen alle Nachfolger vorhanden sind, werden als ALL-Knoten bezeichnet.

Bemerkung: Eine *Strategie* ist ein Teilbaum von G, der eine Schranke für den Minimaxwert von der Wurzel von G beweist. Eine MIN-Strategie beweist eine obere Schranke für den Minimaxwert von G (diese Schranke entspricht gerade dem größten Blattwert der Strategie), und eine MAX-Strategie beweist eine untere Schranke (diese Schranke entspricht gerade dem kleinsten Blattwert der Strategie). In der Literatur findet man auch oft den Begriff *solution tree* anstelle des Strategiebegriffs. □

Definition 1.8-6 **(Blattdisjunkte Strategien)**
Zwei Strategien werden *blattdisjunkt* genannt, wenn sie kein Blatt gemeinsam haben.

Bemerkung: Da Strategien immer bis an die Blätter von Hüllen heranreichen, gibt es keine Blätter einer Strategie, die gleichzeitig innere Knoten anderer Strategien sind. □

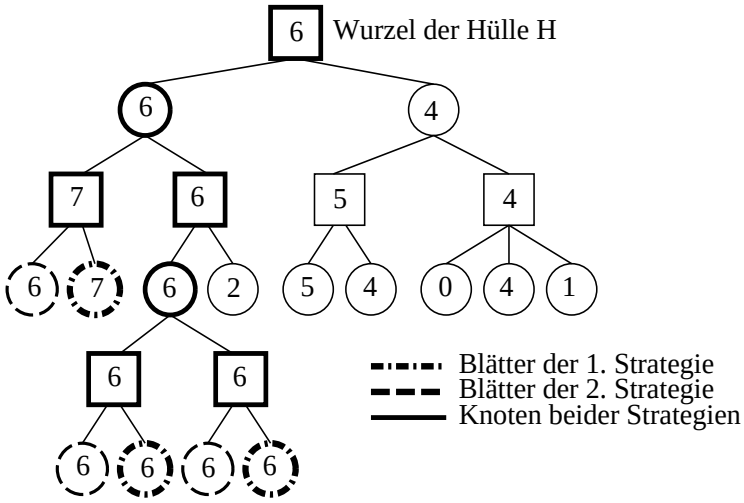


Abb. 1.6: Zwei blattdisjunkte Strategien beweisen an der Wurzel sechs als untere Schranke

Definition 1.8-7 **(Conspiracy Number)**
Es sei H eine Hülle mit Wurzel ϵ. Es gebe eine Strategie, die belegt, daß der Minimaxwert der Hülle $\leq x$ (bzw. $\geq x$) ist. Dann gibt die Conspiracy Number cn(v,x) die kleinste Anzahl von Blättern der Hülle an, die ihren Wert ändern müssen, um den Minimaxwert der Hülle zu einem Wert $> x$ (bzw. $< x$) werden zu lassen. □

1.8.2 Spezielle Definitionen

Definition 1.8-8 **(Bester Zug)**
Sei $H = ((V, E), h)$ eine Hülle. Als einen besten Zug in H bezeichnen wir einen Zug von der Wurzel zu einem Wurzelnachfolger, der denselben Wert hat wie die Wurzel. Sei $m = (v, v')$ so ein Zug. Wir sagen m ist *sicher* mit Conspiracy Number C und der Tiefe d, wenn es ein $x \in \mathbb{Z}$ gibt, so daß es

a) wenigstens C blattdisjunkte Beweisstrategien in H gibt, deren Blätter mindestens in Tiefe d liegen und beweisen, daß der Minimaxwert des besten Zuges größer oder gleich x ist, und

b) für alle anderen Nachfolger der Wurzel gilt, daß es wenigstens C blattdisjunkte Beweisstrategien in H gibt, deren Blätter mindestens in Tiefe d liegen und beweisen, daß der Minimaxwert des jeweiligen Wurzelnachfolgers höchstens x ist.

Auf diese Weise ist sichergestellt, daß sich mindestens C Blätter im Wert ändern müssen, damit sich die Entscheidung an der Wurzel ändert. □

Definition 1.8-9 **(Wert)**
Sei $T = ((V, E), h)$ ein Spielbaum. Ein *Wert* ist ein Tupel

$$w = (a, z) \in \{\text{'}\leq\text{'} , \text{'}\geq\text{'} , \text{'\#'} \} \times \mathbb{Z}.$$

Wir nennen a das *Attribut* des Wertes w und z den *Zahlwert* von w.
$W = \{\text{'}\geq\text{'} , \text{'}\leq\text{'} , \text{'\#'} \} \times \mathbb{Z}$ ist die Menge der möglichen Werte. Wenn wir die Bindung eines Wertes w an einen Knoten v ausdrücken möchten, schreiben wir diesen als $w_v = (a_v, z_v)$. Wenn v_i der i-te Nachfolger eines Knotens v ist und es zu keinen inhaltlichen Verwechslungen kommen kann, erlauben wir uns statt $w_{v_i} = (a_{v_i}, z_{v_i})$ abkürzend $w_i = (a_i, z_i)$ zu schreiben.
Bemerkung: Sei v ein Knoten. $w_v = (\text{'}\leq\text{'} , x)$ wird im folgenden ausdrücken, daß es einen Teilbaum (eine (Teil)-Hülle) unter v gibt, dessen Minimaxwert $\leq x$ ist. $w_v = (\text{'}\geq\text{'} , x)$ wird analog benutzt. $w_v = (\text{'\#'} , x)$ impliziert, daß es eine Teilhülle unter v gibt, deren Minimaxwert $\leq x$ ist, und daß es eine Teilhülle unter v gibt, deren Minimaxwert $\geq x$ ist. Die beiden Hüllen müssen nicht dieselben sein. □

Zwei Werte w_v^1 und w_v^2 (hochgestellte Zahlen sind hier Indizes) an ein und demselben Knoten v können in verschiedenen Relationen zueinander stehen. Uns interessieren in erster Linie die Relationen 'widersprüchlich', 'unterstützend' und 'ungeklärt'.

Definition 1.8-10 **(widersprüchlich, unterstützend, ungeklärt)**
$w^1 = (a^1, z^1)$ ist widersprüchlich zu $w^2 = (a^2, z^2)$ g.d.w. gilt:

($a^1 \in \{$'$\geq$' , '#' $\}$ und $a^2 \in \{$'$\leq$' , '#' $\}$ und $z^1 > z^2$),
oder
es ist ($a^1 \in \{$'$\leq$' , '#' $\}$ und $a^2 \in \{$'$\geq$' , '#' $\}$ und $z^1 < z^2$).

$w^1 = (a^1, z^1)$ unterstützt $w^2 = (a^2, z^2)$ g.d.w. gilt:

($a^1 \in \{$'$\geq$' , '#' $\}$ und $a^2 =$ '$\geq$' und $z^1 \geq z^2$),
oder
es ist ($a^1 \in \{$'$\leq$' , '#' $\}$ und $a^2 =$ '$\leq$' und $z^1 \leq z^2$),
oder
($z^1 = z^2$ und $a^1 = a^2 =$ '#').

$w^1 = (a^1, z^1)$ ist ungeklärt zu $w^2 = (a^2, z^2)$ g.d.w. gilt:

w^1 unterstützt w^2 nicht, und w^1 widerspricht w^2 nicht.

Beispiele
So sind z.B. $w_v^1 =$ ('$\leq$' , 5) und $w_v^2 =$ ('$\geq$' , 6) an einem Knoten v widersprüchlich, da sie zwei Schranken repräsentieren, die innerhalb einer ausgewerteten Hülle nicht beide gelten können. (Trotzdem können sie durchaus an ein und demselben Knoten beobachtet werden, nämlich als Werte zweier unterschiedlicher Hüllen.)
Ist $w_v^1 =$ ('$\geq$' , 5), so widerspricht der Wert $w_v^2 =$ ('$\geq$' , 6) w_v^1 nicht. Er suggeriert zwar eine genauere Schranke, unterstützt aber im wesentlichen den Wert w_v^1.
Den Fall, daß z.B. $w_v^1 =$ ('$\geq$' , 5) und $w_v^2 =$ ('$\leq$' , 6) sind, bezeichnen wir mit 'ungeklärt'. Nehmen wir einmal an, die beiden Werte seien Auswertungen zweier verschiedener Hüllen. Dann wissen wir nicht, ob eine vollständige Auswertung der ersten Hülle der vollständigen Auswertung der zweiten Hülle widerspricht oder ob eine Auswertung die andere unterstützen würde. □

Definition 1.8-11 **(Target)**
Ein *Target* ist ein Tupel $t = (\omega, \delta, \gamma)$, wobei ω ein Wert ist und $\delta, \gamma \in I\!N_0$ sind.

Bemerkung: Targets beschreiben Anforderungen, die wir an Knoten stellen. Sei $t_v = (\omega_v, \delta_v, \gamma_v)$ ein Target, das zu einem Knoten v gehört. δ_v drückt die geforderte Distanz von Blättern der endgültigen Hülle zum Knoten v in der endgültigen Hülle aus. γ_v ist die *Conspiracy Number* von t_v. Sie informiert darüber, auf wieviele blattdisjunkte Beweisstrategien sich das Ergebnis stützen muß. Falls die Anforderungen an die spätere Hülle, die vom Target t_v beschrieben werden, erfüllt worden sind, sprechen wir davon, daß das Target t_v *erfüllt* wurde. □

Kapitel 2

Das Cc2s-Verfahren

2.1 Beschreibung des Cc2s-Algorithmus

Bevor wir in diesem zentralen Kapitel dieser Arbeit den Cc2s-Algorithmus beschreiben (vgl. auch [Lor00]), zeigen wir, daß die Begriffe 'Conspiracy Number' und 'Blattdisjunkte Strategien' eng miteinander verwandt sind und als Synonyme verwendet werden können.

Es folgt eine Metabeschreibung der Cc2s-Suche, die die wichtigsten Ideen und die Vorgehensweise grob erklärt. Der Name Cc2s steht für eine Conspiracy Number Suche, bei der wir einfordern, daß sich wenigstens ein beliebiges Blatt im Wert beliebig ändern darf, ohne die Entscheidung an der Wurzel verändern zu können, also eine Suche mit Conspiracy Number 2. Das erste Beispiel läßt sich dann bereits ohne formale genaue Kenntnis des Vorgehens verstehen.

Wir beschrieben dann den Algorithmus detailliert in top-down Manier. Das Kapitel wird abgerundet durch Überlegungen zur Korrektheit des Verfahrens und durch Analysen des Suchaufwands sowie durch ein weiteres Beispiel.

2.1.1 Conspiracy Numbers und blattdisjunkte Strategien

Satz 2.1-1

Es sei $G = ((V, E), h)$ ein Spielbaum mit Wert $\leq x$ ($\geq x$) an der Wurzel von G. Dann sind äquivalent:

1. Es gibt c blattdisjunkte Min- (Max-)Strategien, die belegen, daß der Wert an der Wurzel $\leq x$ ($\geq x$) ist.

2. Man muß die Werte von mindestens c Blättern ändern, um den Wert der Wurzel zu

$> x$ ($< x$) zu ändern.

□

Beweis :

a) $\rightarrow$ **b)** Es seien $S_1, \ldots, S_c$ c blattdisjunkte Strategien, die belegen, daß der Wert der Wurzel von $G \leq x$ (bzw. $\geq x$) ist. Ändert man nun $c' < c$ Blattwerte, so bleibt auf jeden Fall eine Strategie $S \in \{S_1 \ldots S_c\}$ übrig, die belegt, daß der Wert der Wurzel von $G \leq x$ (bzw. $\geq x$) ist.

b) $\rightarrow$ **a)**

Wir beschreiben im folgenden eine 'Zerstörungsstrategie', die mit der Änderung von c Blattwerten alle c blattdisjunkten Strategien zerstört, die die entsprechende Schranke an der Wurzel belegen.

Induktion über die Tiefe t des Spielbaums:

Sei $t = 0$. Bei einem Spielbaum, der nur aus einem Knoten besteht, ist klar, daß es nur eine Strategie gibt und daß man diese durch die Veränderung eines Wertes zerstören kann.

Die Induktionsvoraussetzung sei also: Unter der Annahme, daß G nur $c' < c$ blattdisjunkte Strategien $S_1 \ldots S_{c'}$ (die x als obere (untere) Schranke belegen) enthält, genügen c' Blattwertänderungen, um den Wert der Wurzel $> x$ (bzw. $< x$) werden zu lassen.

Induktionsschluß: $t - 1 \rightarrow t$

Wir unterscheiden zwei Fälle:

Wenn die Wurzel ϵ von G ein ALL-Knoten ist und sich unter ϵ genau c' belegende Strategien dafür befinden, daß der Wert der Wurzel $\leq x$ (bzw. $\geq x$) ist, befinden sich unter allen Nachfolgern der Wurzel mindestens c' viele solcher belegenden Strategien. Unter mindestens einem der Nachfolger befinden sich aber auch genau c' viele belegende Strategien. Einen dieser letztgenannten Nachfolger wählen wir aus, und mit der Induktionsvoraussetzung folgt die Behauptung.

Wenn die Wurzel ϵ von G ein CUT-Knoten ist und sich unter ϵ genau c' belegende Strategien dafür befinden, daß der Wert der Wurzel $\leq x$ (bzw. $\geq x$) ist, kann das nur daran liegen, daß die Summe $\sum_{i=1}^{d} c'_i$ (d die Anzahl der Nachfolger von ϵ, c'_i die Anzahl der belegenden Strategien des i-ten Nachfolgers v_i der Wurzel ϵ) gleich c' ist. Mit Hilfe der Induktionsvoraussetzung sorgen wir dafür, daß alle diese Nachfolger einen Wert $> x$ (bzw. $< x$) bekommen. Damit wird auch der Wert der Wurzel $> x$ (bzw. $< x$).

2.1.2 Metabeschreibung des neuen Vorgehens

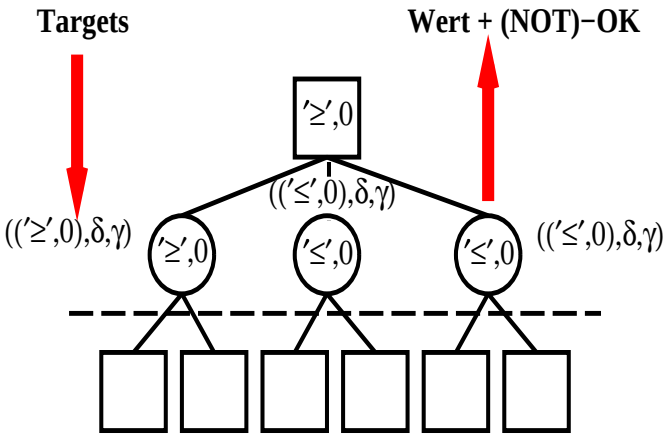


Abb. 2.1: Prinzipielles Verhalten

Abbildung 2.1 verdeutlicht den Datenfluß unseres Algorithmus. Im Gegensatz zum Minimax-Algorithmus oder dem $\alpha\beta$-Algorithmus sind wir nicht gezwungen, den Minimaxwert der Wurzel zu bestimmen. Stattdessen versuchen wir, den besten Zug dadurch von den anderen zu trennen, daß wir einen Spaltwert x bestimmen, von dem wir wissen, daß der Wert des Nachfolgers, der zum besten Zug gehört, mindestens x ist und die Werte der anderen Wurzelnachfolger höchstens x sind.

Ganz zu Anfang werden alle Wurzelnachfolger bewertet. Danach stellt der bis dato besuchte Spielbaum zu jedem Zeitpunkt einen Spaltwert x und einen besten Zug m zur Verfügung. Wie sicher dieser Zug ist, wissen wir allerdings nur zu bestimmten Zeitpunkten. Solange wir nicht wissen, wie sicher m ist, betrachten wir x und m lediglich als Hypothese. Wir beauftragen jeden Nachfolger der Wurzel, entweder seinen Teil der Hypothese zu 'verifizieren' oder zu zeigen, daß neue Bewertungen, die durch eine Erweiterung des Suchbaums zustande gekommen sind, die Hypothese 'falsifizieren'. Die Begriffe 'verifizieren' und 'falsifizieren' werden von uns auf eine weiche Art verwendet. Sie beziehen sich immer auf den neuesten Stand des Wissens und nicht auf absolute Wahrheit. Eine 'Verifikation' ist also nicht mit dem Beweis vergleichbar, den ein $\alpha\beta$-Algorithmus für eine Schranke des Wurzelwertes durchführt. 'Verifikation' bedeutet eher, Vertrauen in einen Wert zu stärken. Die Verifikation wird mit Hilfe von Targets durchgeführt, die von der Wurzel zu den Blättern aufgeteilt und verteilt werden. Jeder Knoten v, der ein Target bekommen hat, nimmt seinen eigenen Wert als richtig an, auch wenn unter v ein paar Knoten dazugefügt werden. Er bildet für seine Nachfolger Unterhypothesen und Teiltargets in der Art, daß wenn die Nachfolger ihre Teiltargets erfüllen, auch v selber sein

Target erfüllt. Dann beauftragt v seine Nachfolger, die entsprechenden Unterhypothesen zu untersuchen. Wenn ein Knoten v sein Target erfüllen konnte, sendet es ein OK an seinen Vorgänger. Wenn das nicht möglich war, hat sich der aktuelle Wert von v geändert. In diesem Fall meldet v den neuen Wert und ein NOT-OK an seinen Vorgänger.

2.1.3 Ein erstes Beispiel

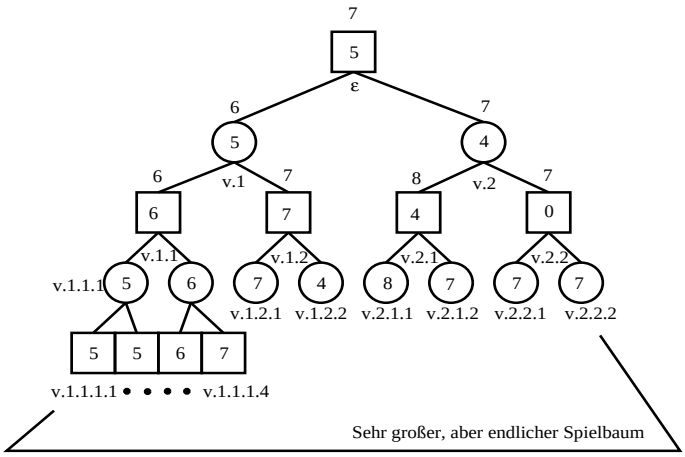


Abb. 2.2: Beispiel-Spielbaum

Abb. 2.2 zeigt die obere Spitze eines Spielbaums, der im folgenden Grundlage unserer Beispiele ist. Sei h die zu diesem Spielbaum gehörige heuristische Bewertungsfunktion. Wir gehen im folgenden davon aus, daß wir eine Bewertungs-Prozedur zur Verfügung haben, die uns entweder die heuristische Zahl $h(v)$ als Funktionswert eines Knotens v zurückliefert (das führt z.B. bei Knoten $v.2.1$ zu dem von uns so genannten 'Punktwert' $(\text{'\#'}, 4)$), oder die Frage beantwortet, ob $h(v)$ kleiner oder gleich einer Schranke x ist. Im letzteren Fall wäre die Antwort unserer Bewertungs-Prozedur auf die Frage, ob der Wert von $v.2.1$ kleiner oder gleich 5 ist somit $(\text{'}\leq\text{'}, 5)$. Wir nennen so einen Wert im folgenden auch einen 'Schrankenwert'.

An den inneren Knoten des Beispiel-Spielbaums befinden sich zwei Zahlen: Die innere ist eine punktgenaue, direkte heuristische Bewertung $h(\dots)$. Über den Knoten befinden sich die Minmaxwerte, die sich aus den heuristischen 'Werten' (genauer: Zahlen; nicht zu verwechseln mit unserer speziellen Definition von 'Wert') der 'Blätter' ergeben. Blätter sind hierbei die Knoten $v.1.1.1.1$ bis $v.1.1.1.4$, $v.1.2.1$, $v.1.2.2$, $v.2.1.1$, $v.2.1.2$, $v.2.2.1$ und $v.2.2.2$. Unter den Knoten befinden sich die Namen der Knoten, zugewiesen nach dem Dewey-Dezimal System.

Im folgenden sehr einfachen ersten Beispiel, in dem jede Erweiterung des Suchbaums zu den gefaßten Hypothesen paßt, ist es die Aufgabe, einen besten Zug zu finden, der sich auf eine Hülle stützt, deren Blätter eine Entfernung von zwei zur Wurzel haben. Die Abbildungen 2.3-2.5 zeigen das inkrementelle Wachstum der Strategien.

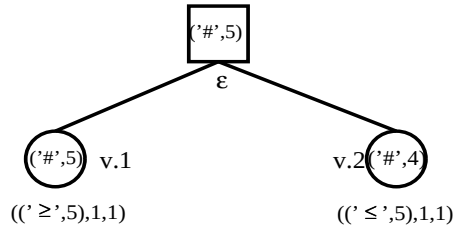


Abb. 2.3: Ausgangssituation

Zunächst werden alle Nachfolger der Wurzel ϵ mit einem Punktwert versehen, um überhaupt eine Starthypothese zu bekommen. Da uns unsere heuristische Bewertung für die Knoten $v.1$ und $v.2$ Punktwerte geliefert haben soll, bekommen die Werte das Attribut '#'. Das bedeutet ja z.B. für $v.1$ nichts anderes, als daß es unter $v.1$ eine Hülle gibt, die einen Minimaxwert von ≥ 5 besitzt, und daß es eine (möglicherweise andere) Hülle gibt, deren Minimaxwert ≤ 5 ist. In diesem Fall bestehen beide Hüllen nur aus $v.1$.

Bezüglich der durchgeführten Tiefe-1 Suche ist $v.1$ der beste Wurzelnachfolger (d.h. der mit dem höchsten Wert). Deshalb bilden wir die Hypothese, daß er auch bei einer Tiefe-2 Suche der beste Zug bleiben wird. Für $t_{v.1}$ bilden wir das Target $t_{v.1} = (('\geq' , 5), 1, 1)$, und für $v.2$ bilden wir $t_{v.2} = (('\leq' , 5), 1, 1)$. (s. Abb. 2.3)

An $v.1$ beginnen wir mit dem Verifikationsprozeß, da es der linke Nachfolger der Wurzel ist. Es werden solange Nachfolger generiert, wie sie alle einen Wert ≥ 5 haben (denn um das Target zu erfüllen, müssen alle einen solchen Wert vorweisen).

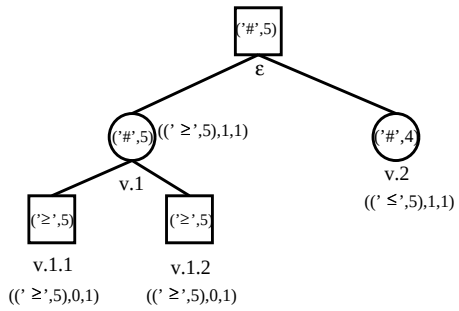


Abb. 2.4: Nach der Expansion des Knotens v_1

Auch den Knoten $v.1.1$ und $v.1.2$ werden Targets zugewiesen (Abb. 2.4). Der Algorithmus

untersucht diese Knoten der Reihe nach und gibt jeweils ein OK zurück, da die Targets trivialerweise erfüllt sind. (Es wurde nämlich gefordert, daß die Werte von $v.1.1$ und $v.1.2$ sicher sind mit einer Resttiefe von 0 und einer Conspiracy Number von 1.) Danach gibt auch $v.1$ ein OK an die Wurzel zurück.

Anschließend wird der erste Nachfolger von $v.2$ generiert, und es wird die Bewertung angefragt, ob der Wert dieses neuen Nachfolgers ≤ 5 ist. Das ist der Fall (Abb. 2.5), und so geben auch $v.2.1$ an $v.2$ und $v.2$ an ϵ jeweils OK zurück. Wir wissen nun, daß der Zug

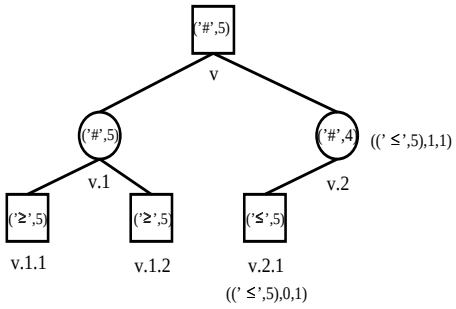


Abb. 2.5: Nach der Expansion des Knotens v_2

von der Wurzel nach $v.1$ sicher ist mit Tiefe 2 und Conspiracy Number 1. Die Werte der inneren Knoten wurden nur gebraucht, um die Suche zu steuern.

Die Targets benötigen wesentlich weniger Platz als die Conspiracy Number Vektoren der konventionellen CNS. Darüberhinaus erlauben sie einen effizienten Beschneidungsmechanismus, ähnlich dem des $\alpha\beta$-Algorithmus. Wir können zusätzlich ausnutzen, daß es oft mehr Zeit kostet, einen punktgenauen heuristischen Wert auszurechnen, als einfach nur in Erfahrung zu bringen, ob dieser Wert größer oder kleiner einer vorgegeben Zahl ist. Diese Eigenschaften ermöglichen es dem Cc2s-Algorithmus, auch für positionelle Suchen eingesetzt zu werden, für die man eine feingranulare Bewertung braucht. Das top-down Verteilen von Targets bietet eine hohe Flexibilität und erlaubt es, nahezu beliebige Sicherheitskonzepte zu verwirklichen. Im folgenden präsentieren wir eine hybride Mischung aus Suchen mit fester Tiefe (das sind Suchen, bei denen alle Blätter die gleiche Entfernung zur Wurzel haben) und Conspiracy Number Suchen.

2.1.4 Algorithmische Details

Der Suchalgorithmus

Um den Suchalgorithmus übersichtlicher darstellen zu können, haben wir ihn unterteilt. Es gibt zum einen die Prozedur, die die Entscheidungsauswahl an der Wurzel des Spielbaums vornimmt, und die rekursive Prozedur Cc2s(. . .), die sich nicht mehr um die Aus-

wahl eines Zuges kümmern muß, sondern nur noch Schranken von Werten nachweist. Diejenige Prozedur, die an der Wurzelstellung eine Entscheidung extrahiert und die den Startpunkt unseres Algorithmus darstellt, nennen wir DetermineMove(root ϵ, Resttiefe d). d steht dabei für die vom Benutzer geforderte Tiefe. Die geforderte Conspiracy Number C wird in der Prozedur DetermineMove(...) in Zeile 5 festgelegt.

```
DetermineMove(root ε /* oBdA sei ε ein MAX-Knoten */), Resttiefe d)
1       Erzeuge alle Nachfolger der Wurzelstellung ε, i.e. Γ(ε);
2           /* Seien v_1 ... v_b die Nachfolger von ε,
3               t_1 ... t_b die Targets der Knoten v_1 ... v_b, und
4               w_1 ... w_b die Werte, die zu den Knoten v_1 ... v_b gehören. */
5       C := 2; /* C := 1 führt zu herkömmlichen Suchen fester Tiefe. */
6       for all v_i ∈ Γ(ε) w_i := Evaluate(v_i, −∞, ∞);
7       w_ε := ('#' , max{z_i; i ∈ {1...b}});
8           /* Alle Nachfolger v_i der Wurzel haben einen Wert
9               der Form w_i = ('#' , z_i). */
10      do {
11          tausche v_1 und v_j so, daß danach z_1 ≥ z_i, ∀i ∈ {1...b};
12          t_1 := (('≥' , z_1), d − 1, C);
13          for all i ∈ {2...b} t_i := (('≤' , z_1), d − 1, C);
14          for i := 1 to |Γ(ε)| do {
15              r := Cc2s(v_i, t_i);
16              w_ε := UpdateValue(ε);
17              if r = NOT_OK break ;
18                  /* Verlasse die for-Schleife und stelle neue Hypothesen auf! */
19          }
20      } while r = NOT_OK;
```

Abb. 2.6: Entscheidungsfindung auf oberster Ebene

Falls die Nachfolger der Wurzel noch nicht generiert worden sind, werden sie erzeugt und mit Werten der Form $('\#' , \ldots)$ versehen (Abb. 2.6, Z. 1-9). Es handelt sich dabei um Punktwerte, die wie oben beschrieben von der Bewertungsfunktion geliefert werden können. Die Routine DetermineMove(...) stellt dann auf Grundlage des aktuellen Spielbaums eine Hypothese über den besten Zug auf (Abb. 2.6, Z. 11-13). Das ist der Zug, der den augenblicklich höchsten Minimaxwert besitzt. Da der aktuelle Spielbaum zumindest die Wurzel selber und alle ihre Nachfolger enthält (zumindest nach Beendigung der Zeile 7), ist dies immer möglich. Der beste Nachfolger der Wurzel wird mit dem Target $(('\geq' , z_1), d-1, C)$ versehen und alle anderen Nachfolger mit Targets $(('\leq' , z_1), d-1, C)$. Dann wird die Prozedur Cc2s(...) beauftragt, die Verifika-

tion durchzuführen. DetermineMove(...) wiederholt die beschriebenen Schritte, bis alle Wurzelnachfolger mit OK geantwortet haben.

Seien $\Gamma'(v)$ die Nachfolger eines Knotens v, die im aktuellen Suchbaum vorhanden sind. Sei t_v das Target für v, w_v der Wert von v. Seien $v_1 \dots v_{|\Gamma'(v)|}$ die Nachfolger von v, bezüglich des aktuellen Suchbaums, $t_1 \dots t_{|\Gamma'(v)|}$ die Targets der Knoten $v_1 \dots v_{|\Gamma'(v)|}$, und $w_1 \dots w_{|\Gamma'(v)|}$ ihre Werte.

Wir bezeichnen einen Knoten als OnTarget(v, t_v), wenn der Wert von v nicht im Widerspruch zu der Wertkomponente des Targets t_v steht (vgl. Def. 'Wert' und 'widersprüchlich', S.22). Dieses Attribut eines Knotens bringt zum Ausdruck, daß sich der Sucher noch auf dem richtigen Weg befindet. Wenn die Prozedur Cc2s(...) (Abbildung

```
bool Cc2s(node v, target t_v = ((α_v, β_v), δ_v, γ_v))
1       if (δ_v = 0 and γ_v ≤ 1) or |Γ(v)| = 0 return OK;
2       r := NOT_OK;
3       while r = NOT_OK do {
4           PartialExpansion(v, t_v);
5           if not OnTarget(v, t_v) return NOT_OK;
6           Split(v, t_v, v_1 ... v_|Γ'(v)|); /* weist den Nachfolgern Targets zu */
7           for i := 1 to |Γ(v)'| do {
8               r := Cc2s(v_i, t_i);
9               w_v := UpdateValue(v);
10              if not OnTarget(v, t_v) return NOT_OK;
11              if r = NOT_OK break ; /* die for-Schleife verlassen, springe zu Z.3 */
12          }
13      } /* while ... */;
14      return OK;
```

Abb. 2.7: Rekursive Suchprozedur

2.7) für einen Knoten v aufgerufen wird, ist garantiert, daß der Wert von v die Wertkomponente des Targets t_v unterstützt (entweder durch DetermineMove(...) oder wegen Abbildung 2.7, Z. 4-6). Zu allererst wird kontrolliert, ob v ein Blatt ist, d.h., ob t_v trivial erfüllt ist (Z.1). Das ist der Fall, wenn die erforderliche Suchtiefe erreicht worden ist ($\delta_v = 0$) und wenn die geforderte Anzahl blattdisjunkter Beweisstrategien Eins ist ($\gamma_v \leq 1$). Ein Knoten v ist auch dann ein Blatt, wenn v ein Blatt des Gesamtspielbaums ist. Das zugrunde liegende Spiel läßt dann eine Erweiterung des Knotens v nicht zu (z.B. wegen eines Schachmatts).

Der Unteralgorithmus PartialExpansion(...) versucht, Nachfolger zu finden, die für die sogenannte Split-Operation gut geeignet sind. Er bewertet von links nach rechts Nach-

folger von v, die entweder noch nie bewertet worden sind, oder von denen 'ungeklärt' ist, ob ihre Werte im Widerspruch zur Wertkomponente des Targets t_v stehen (s. Def. 'Wert', 'widersprüchlich' und 'ungeklärt', S.22). Wenn ein Knoten s schon einmal bewertet worden ist und nun noch einmal bewertet wird, wird er mit einem Punktwert der Form $('\#' , z_s)$ versehen. Es ist wichtig, daß ein '#' -Wert w_v nie ungeklärt in Relation zu der Wertkomponente eines ebenfalls zu v gehörenden Targets t_v steht. Deshalb kann jeder Knoten höchstens zweimal bewertet werden!

Für einen noch nie zuvor bewerteten Nachfolger s von v wird angefragt, ob der Wert von s die Wertkomponente des Targets t_v unterstützt oder ihr widerspricht. s bekommt dann einen Wert $('\geq' , z_s)$ oder $('\leq' , z_s)$. Der Unteralgorithmus zur Teilexpansion geht von links nach rechts vor und ruft nach jeder Bewertung auch für den Knoten v den UpdateValue-Operator auf.

Falls v ein ALL-Knoten ist und eine Teilerweiterung des Suchbaums eine Veränderung des Wertes von v hervorbringt, so daß dieser neue Wert dem Target t_v widerspricht, hört PartialExpansion(...) sofort damit auf, Nachfolger von v zu bewerten und Cc2s(...) verläßt v mit Hilfe von Zeile 5. Wenn v ein CUT-Knoten ist, bewertet PartialExpansion(...) solange Nachfolger von v, bis es γ_v-viele geeignete gefunden hat, auf die sich das Target t_v verteilen läßt, oder bis alle Nachfolger von v bewertet worden sind.

Anschließend wird das Target t_v 'gesplittet', d.h., es werden Targets für die Nachfolger von v erarbeitet (Zeile 6). Das Aufteilen von Targets wird von der Prozedur Split(...) so durchgeführt, daß das Target von v erfüllt sein wird, wenn die neu gebildeten Teiltargets erfüllt werden. Split(...) benötigt die von der Prozedur PartialExpansion(...) herausgesuchten Kandidaten, auf die das Target t_v aufgeteilt werden kann. In den nächsten Abschnitten werden diese Routinen noch genauer untersucht.

Die neuen Targets werden den Nachfolgern von v zugewiesen, und Cc2s(...) untersucht die Nachfolger von v, bis entweder alle Nachfolger ihre Targets erfüllt haben, oder bis der Wert von v dem Target t_v widerspricht, was bedeutet, daß v nicht mehr OnTarget(...) ist und damit Cc2s(...) sich nicht mehr auf dem richtigen Weg befindet. Wenn ein Aufruf von Cc2s(...) mit dem Resultat OK vom Knoten v_i (Zeile 8) zu v zurückkehrt, konnte v_i sein Target erfüllen. Kehrt Cc2s mit NOT-OK zurück, haben sich Werte unterhalb von v_i verändert, so daß das Target von v_i nicht mehr erfüllbar erscheint. In diesem Fall muß die Prozedur Cc2s(...) entscheiden, ob v ein NOT-OK an seinen Vorgänger melden muß (Z.10) oder ob am Knoten v umdisponiert werden kann und neue Teiltargets für die Nachfolger von v gebildet werden können (Z.11 und 13).

Wertbildung von unten nach oben

Ein wichtiger zu klärender Punkt ist, was geschehen soll, wenn eine Erweiterung des Suchbaums zu einem Widerspruch zwischen einem Wert w_v eines Knotens v und der Wertkomponente eines zu v gehörenden Targets t_v führt. Die Aufgabe ist es dann, die neue Information aufzunehmen und sicherzustellen, daß die Cc2s-Prozedur, die von diesen Werten gesteuert wird, nicht in eine Endlosschleife geschickt werden kann. Wir haben den Operator in Abbildung 2.8 erarbeitet, der uns dabei entscheidend hilft.

```
value UpdateValue(node v)
    ...
1   /* Sei v ein MAX-Knoten */
2   if Γ'(v) ≠ Γ(v) or ∃s ∈ Γ'(v) with a_s = '≥' or
3       ∃s ∈ Γ'(v) with (a_s = '≤' and z_s ≥ z_v) then {
4           if a_v = '≤' and ∃s ∈ Γ'(v) with (a_s ∈ {'≥' , '#' } and z_s ≥ z_v)
5                   then a_v := '#' ;
6           z_v := max{z_v, max{z_s | s ∈ Γ'(v), a_s ∈ {'#' , '≥' }};
7   } else {
8       if a_v = '≥' or (a_v = '≤' and ∃s ∈ Γ'(v) with (a_s = '#' and z_s ≥ z_v))
9           then      a_v := '#' ;
10      z_v := max{z_s | s ∈ Γ'(v)};
    }   /* Falls v ein MIN-Knoten ist, ist das Ergebnis analog definiert. */
        /* Man tausche ≤ mit ≥, und max mit min. */
```

Abb. 2.8: Update von heuristischen Knotenwerten

Vereinfacht gesagt, macht UpdateValue(...) folgendes: Es nimmt sich die Werte der Nachfolger eines Knotens v vor, bildet so weit wie möglich Minimaxinformationen daraus und reichert den entstandenen Wert mit den alten Informationen, dem alten Wert von v an.

Betrachten wir zwei Beispiele eines Widerspruchs und seiner Auflösung: Sei v ein MAX-Knoten mit dem Wert $('\leq' , 5)$. Der Wert soll dadurch zustande gekommen sein, daß alle Nachfolger bewertet wurden und einen Wert $('\leq' , 5)$ haben. Nun bewerten wir v mit Hilfe unserer Bewertungsprozedur v direkt und erhalten einen Wert > 5, z.B. $('\geq' , 7)$ oder $('\#' , 7)$. So, wie wir den Cc2s-Algorithmus bisher beschrieben haben, kann der Fall, daß die Bewertungsprozedur $('\geq' , 7)$ liefert zwar streng genommen nicht eintreten. Trotzdem halten wir es für sinnvoll, auch solche Fälle mit abzudecken. Dadurch ist es zum einen möglich, Beschleunigungsheuristiken wie in Abschnitt 2.3 einzusetzen. Zum anderen kann es in der Praxis Sinn machen, eine heuristische Bewertung zu verwenden, die bei mehrfacher Bewertung eines Knotens nicht unbedingt jedesmal den gleichen Wert

zurückliefert. UpdateValue weist v den Wert $(\text{'\#'}, 5)$ (Abb. 2.9) zu und löst somit den

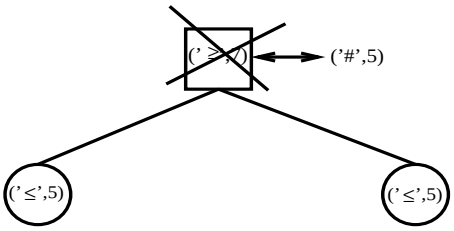


Abb. 2.9: Widerspruchsauflösung (1)

Widerspruch auf (Abb. 2.8,Z.9,10). Um den Sinn dieser Operation zu verstehen, müssen wir uns in Erinnerung rufen, daß die Werte innerer Knoten lediglich zur Steuerung der Cc2s-Routine dienen. Daß das Ergebnis an der Wurzel, der beste Zug, auch wirklich vertrauenswürdig ist, wissen wir nur durch die Werte der Blätter und durch den Target-Mechanismus. Insofern ist ein innerer Wert ein heuristischer Helfer. Nun besagte der alte Wert, daß es eine Hülle H_1 unterhalb von v gibt, die einen Minimaxwert von ≤ 5 hat. Die direkte Bewertung besagt, daß es eine Hülle H_2 gibt, deren Minimaxwert ≥ 7 ist. Die Hülle besteht nur aus v selber. Offenbar ist dann doch die Hülle H_2 eine echte Teilhülle von H_1. Wenn man davon ausgeht, daß mehr Suche zu besseren Ergebnissen führt (und alles andere wäre absurd, da man dann jedwede Suche besser bleiben ließe), macht es hier keinen Sinn, davon auszugehen, daß man ein Target mit einer Wertkomponente $(\text{'}\leq\text{'}, 5)$ erfüllen kann. Andererseits sagt die Hülle H_1 nichts über eine untere Schranke des Wertes von v aus. Da es eine Hülle gibt, die eine untere Schranke von 5 plausibel erscheinen läßt (die Hülle H_2), ist der Knoten immer noch ein guter Kanditat, wenn es darum geht, ein Target mit Wertkomponente $(\text{'}\geq\text{'}, 5)$ zu erfüllen.

Ein anderes Beispiel: Sei v ein MAX-Knoten mit dem Wert $(\text{'}\geq\text{'}, 5)$ und zwei Nachfolgern. Einer dieser Nachfolger habe den Wert $(\text{'}\leq\text{'}, 3)$ bekommen. Falls nun der zweite Nachfolger einen Wert $(\text{'}\leq\text{'}, 7)$ (Abb. 2.10c), oder $(\text{'}\geq\text{'}, 3)$ (Abb. 2.10d) zugewiesen bekommt oder gar nicht bewertet worden ist (Abb. 2.10b), bleibt der Wert von v unverändert (wegen Z.6). Wenn jedoch der zweite Nachfolger den Wert $(\text{'}\leq\text{'}, 3)$ bekommt (Abb. 2.10a), wird der neue Wert von v mit Hilfe der Zeilen 9 und 10 ermittelt, und v bekommt den Wert $(\text{'\#'}, 3)$.

Andere UpdateValue Operationen sind intuitiv einfach nachzuvollziehen: Sei z.B. der Wert eines MAX-Knotens v $(\text{'}\geq\text{'}, 5)$. Einer der Nachfolger bringe aber einen Wert $(\text{'}\geq\text{'}, 6)$ ein. Dann bekommt auch v den Wert $(\text{'}\geq\text{'}, 6)$.

Es gibt ein paar Eigenschaften der so erzeugten Werte, die die so gebildeten Werte zu mehr als einer willkürlichen Heuristik machen und Korrektheits- und Terminierungsargumentationen vereinfachen.

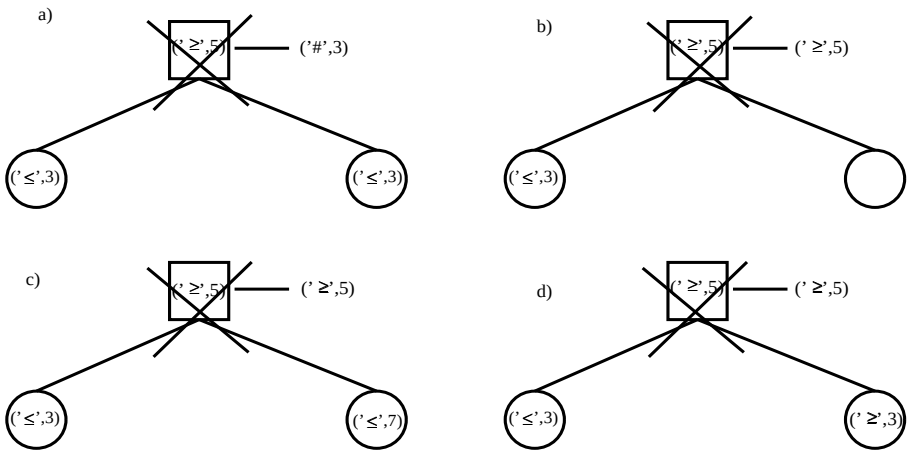


Abb. 2.10: Widerspruchsauflösung (2)

Satz 2.1-2 **(Eigenschaften des Update-Operators)**
Wenn alle inneren Knotenwerte, von den Blättern startend zur Wurzel hin, mit Hilfe von UpdateValue gebildet werden, können wir folgendes beweisen:

(a) Aus $w_v = (\text{'}\leq\text{'}\ , x)$ folgt, daß es eine Hülle unter v gibt, deren Minimaxwert $\leq x$ ist, **(b)** Aus $w_v = (\text{'}\geq\text{'}\ , x)$ folgt, daß es eine Hülle unter v gibt, deren Minimaxwert $\geq x$ ist, und **(c)** $w_v = (\text{'\#'}\ , x)$ impliziert, daß es eine Hülle unter v gibt, deren Wert $\leq x$ ist, und daß es eine (möglicherweise ganz andere) Hülle unter v gibt, deren Wert $\geq x$ ist. Die beiden Hüllen sind nicht notwendigerweise identisch.

Darüberhinaus gelten dann folgende wichtige Eigenschaften: **(d)** Sei w_t die Wertkomponente eines Targets t, die von w_v (einem Wert eines Knotens v) unterstützt wird. Wenn v durch t zu einem ALL-Knoten wird, widerspricht kein Wert der Nachfolger von v dem Wert w_t. Wenn v durch t zu einem CUT-Knoten wird, gibt es einen Nachfolger von v, dessen Wert dem Wert w_t nicht widerspricht. □

Mit Hilfe der ersten drei Eigenschaften ist es möglich zu zeigen, daß Ergebnisse des Cc2s-Algorithmus letztendlich immer auf Hüllen und deren Minimaxwerten basieren. Aufgrund der letzten beiden Eigenschaften wird folgendes gesichert: Wenn die Prozedur Cc2s(. . .) einen Knoten v besucht, gibt es entweder noch Nachfolger von v, von denen nicht geklärt ist, ob sie dazu beitragen können (dann werden solche Nachfolger bewertet; Abb. 2.7,Z.4), t_v auf die Nachfolger von v zu verteilen, oder die Prozedur Split(. . .) (Abb. 2.7,Z.6) kann das Target t_v, wie im nächsten Abschnitt beschrieben, auf die Nachfolger von v aufteilen. Es kann nicht passieren, daß die Prozedur Cc2s(. . .) aufgrund von Zei-

le 5 verlassen wird, ohne daß vorher ein Knoten bewertet wurde. Damit ist es möglich zu zeigen, daß der Cc2s-Algorithmus nicht in eine Endlosschleife laufen kann, ohne den Suchbaum zu verändern.

Die Form einer Hülle und die Handhabung von Sicherheit

Um an einem MAX-Knoten v eine untere Schranke für seinen Wert zu bestimmen (bzw. um eine obere Schranke an einem MIN-Knoten zu zeigen) braucht man lediglich einen Nachfolger von v zu finden, dessen Wert die Schranke einhält. So einen Knoten nennen wir einen CUT-Knoten (engl. cutnode). Wenn man eine obere Schranke an einem MAX-Knoten v beweisen möchte, müssen alle Nachfolger von v diese Schranke einhalten und einen Minimaxwert haben, der höchstens so groß ist wie der von v. Deshalb nennen wir so einen Knoten ALL-Knoten. Die Targets bestimmen, ob ein Knoten als ALL-Knoten oder als CUT-knoten zu betrachten ist (Abb. 2.11).

bool Allnode(node v, target $t_v = ((\alpha_v, \beta_v), \delta_v, \gamma_v)$)
if v ist ein MAX-Knoten **and** $\alpha_v = \text{'}\leq\text{'}$ **then return** true;
else if v ist ein MIN-Knoten **and** $\alpha_v = \text{'}\geq\text{'}$ **then return** true;
else return false;

bool Cutnode(node v, target $t_v = ((\alpha_v, \beta_v), \delta_v, \gamma_v)$)
return ¬Allnode(v, t_v);

Abb. 2.11: ALL- und CUT-Knoten

Wenn z.B. v ein MAX-Knoten ist und das Target einen Wert der Form $(\text{'}\geq\text{'}, x)$ fordert, wird v zu einem CUT-Knoten.

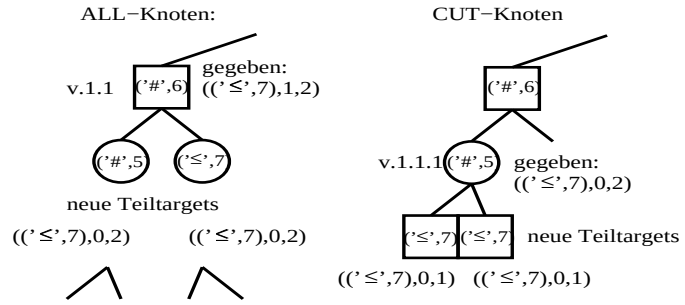


Abb. 2.12: Aufteilung von Targets an CUT- bzw. ALL-Knoten

Wenn v ein Knoten mit Target t_v ist, definieren wir Teiltargets für die Nachfolger von v mit Hilfe folgender Beobachtung: ObdA sei v ein MAX-Knoten. Sei x eine geforderte

obere Schranke des Minimaxwertes von v bezüglich einer Hülle, die durch t_v näher beschrieben wird. Dann muß x eine obere Schranke für die Minimaxwerte aller Nachfolger $v_i, i \in \{1 \dots b_v\}$ von v sein. Falls es C blattdisjunkte Strategien unter jedem einzelnen Sohn von v gibt, die die Schranke x beweisen, so gibt es auch C blattdisjunkte Beweisstrategien für den Minimaxwert x bei v. Allgemein gilt: Wenn C_i die Anzahl blattdisjunkter Beweisstrategien unter den Knoten v_i (für alle Nachfolger v_i von v) ist, dann ist die Anzahl blattdisjunkter Beweisstrategien (die alle die Schranke x zeigen) unter v gleich $\min_{i=1}^{b_v} C_i$.

Jetzt betrachten wir einen MAX-Knoten v, für dessen Minimaxwert x eine untere Schranke ist. Sei c die Anzahl derjenigen Nachfolger von v, die ebenfalls einen Minimaxwert $\geq x$ haben. (Wegen der Minimaxregel gibt es mindestens einen solchen Nachfolger.) Sei $C_i, i \in \{1 \dots c\}$ die Anzahl der blattdisjunkten Beweisstrategien unter v_i, die die Schranke x am Knoten v_i beweisen. Dann ist die Anzahl der Beweisstrategien unter dem Knoten v, die die Schranke x am Knoten v beweisen, gleich der Summe $\sum_{i=1}^{c} C_i$.

Split(node v, target $t = ((\alpha, \beta), \delta, \gamma)$, nodes $v_1 \dots v_{|\Gamma'(v)|}$ /* Nachfolger von v*/)

1 **if** Cutnode(v, t) **then** {
2 /* Seien $w_1 = (a_1, z_1) \dots w_{|\Gamma'(v)|} = (a_{|\Gamma'(v)|}, z_{|\Gamma'(v)|})$ die Werte der
3 Knoten $v_1 \dots v_{|\Gamma'(v)|}$. Wenn diese Zeile erreicht wird, gibt es mindestens
4 einen Index i mit $z_i \geq \beta$ und $a_i \in \{'\geq', '\#'\}$, falls v ein MAX-Knoten ist,
5 bzw. ein i mit $z_i \leq \beta$ und $a_i \in \{'\leq', '\#'\}$, falls v ein MIN-Knoten ist.
6 Falls es noch einen weiteren solchen Index gibt, sei k der kleinste dieser
7 Indizes mit $k \neq i$. Andernfalls setzen wir $k := -1$. */
8 **if** $k = -1$ **or** $\gamma \leq 1$ **then** {
9 $t_i := ((\alpha, \beta), \delta - 1, \gamma)$; /* t_i sei das Target von v_i. */
10 $t_j := ((\alpha, \beta), 0, 0)$ $\forall\, j \in \{1 \dots |\Gamma'(v)|\}$ with $i \neq j$;
11 } **else** {
12 $t_i := t_k := ((\alpha, \beta), \delta - 1, 1)$;
13 $t_j := ((\alpha, \beta), 0, 0)$ $\forall\, j$ mit $i \neq j$ und $k \neq j$;
14 }
15 } **else** $t_i := ((\alpha, \beta), \delta - 1, \gamma)$ $\forall i \in \{1 \dots b\}$
16 /* t_i sei dabei das Target des Knotens v_i */

Abb. 2.13: Die Splitprozedur

Schließlich gilt noch, daß es unter v genau dann eine Strategie gibt, die eine Schranke x beweist und deren Blätter eine Entfernung von d zu v haben, wenn es entsprechende Strategien unterhalb der Nachfolger von v gibt, deren Blätter eine Entfernung von $d - 1$ zu den Nachfolgern von v haben.

Wenn unser Algorithmus einen Knoten v mit Target t besucht, wird zunächst entschieden, ob v als CUT-Knoten oder als ALL-Knoten angesehen werden muß. Dann erzeugt er Teiltargets für die Nachfolger von v in der Art, daß das Target von v erfüllt ist, wenn alle Teiltargets aller Nachfolger von v erfüllt sind (Abb. 2.13).

Wenn ein Target an einem CUT-Knoten mehr als eine blattdisjunkte Beweisstrategie fordert, versuchen wir diese Aufgabe auf zwei geeignete Nachfolger zu verteilen (Abb. 2.13, Z. 12). Cc2s(. . .) verwaltet Sicherheitsinformationen allein durch die Targets, die top-down im Suchbaum verteilt werden.

Mit Hilfe der hier vorgestellten Prozedur Split(. . .) kann die Prozedur Cc2s(. . .) (s. Abb. 2.7) sicher sein, daß ein Knoten v selber sein Target erfüllt hat, wenn alle Nachfolger von v ihre Teiltargets erfüllt haben.

Anstatt in den Zeilen 12 und 13 (Abb. 2.13) die Anforderung von zwei blattdisjunkten Strategien auf den i-ten und k-ten Nachfolger aufzuteilen, kann man natürlich auch leicht höhere Conspiracy Numbers aufteilen, auch auf möglicherweise mehr als zwei stützende Nachfolger von v.

Expansionen und Bewertungen

Zu einer vollständigen Beschreibung unseres neuen Algorithmus fehlen noch die Beschreibungen der Prozeduren PartialExpansion(. . .), Evaluate(. . .) und OnTarget(. . .).

Eine Teilerweiterung des Spielbaums (PartialExpansion) ist die Erzeugung und Bewertung eines einzelnen Knotens.

Wenn v ein CUT-Knoten sein soll (aufgrund des Wertes w_v und des Targets t_v), muß wegen der Definition einer Beweisstrategie mindestens ein Nachfolger den Wert von v stützen. Wenn also PartialExpansion(. . .) bereits γ_v-viele solcher Nachfolger gefunden hat, werden keine weiteren Knoten unter v erzeugt und bewertet (Abb. 2.14, Z. 24). Wenn scheinbar keiner der neu erzeugten Nachfolger von v dabei helfen kann, das Target t_v zu erfüllen, ändert sich der Wert von v (Z. 23). Um in diesem Fall einen möglichst genauen Eindruck davon zu erhalten, wo der Wert von v anzusiedeln ist, werden die Nachfolger von v mit exakten Werten versehen und gegebenenfalls noch einmal bewertet (Z. 27-31). Zum einen führt dieses Vorgehen zu genauerem Wissen über heuristische Werte an Knoten, an denen Bewertungen zu unvorhergesehenen Veränderungen innerhalb des Suchbaums geführt haben, zum anderen vereinfacht es die Analyse des Cc2s-Algorithmus.

Wenn v ein ALL-Knoten ist, müssen alle Nachfolger den Wert von v stützen. Wenn es dazu kommt, daß ein Wert eines neu generierten Nachfolgers dem Wert w_v von v widerspricht, werden keine weiteren Nachfolger unterhalb von v erzeugt oder bewertet (Z. 11-14).

```
PartialExpansion(node v, target t_v = ((α_v, β_v), δ_v, γ_v))
1       /* Seien v_1 ... v_b die Nachfolger von v
2           w_1 ... w_b die Werte der Knoten v_1 ... v_b */
3   if Allnode(v, t_v) then {
4       for i := 1 to b do {
5           if v_i ∉ Γ'(v) then Γ'(v) := Γ'(v) ∪ {v_i};
6           if ExaminationUseful(v_i, t_v) then {
7               if v_i ist noch nie bewertet worden then {
                    if v ist MAX-Knoten then w_i := Evaluate(v_i, β_v, β_v + 1);
                    else w_i := Evaluate(v_i, β_v − 1, β_v);
8               } else w_i := Evaluate(v_i, −∞, ∞);
9           }
10          w_v := UpdateValue(v);
11          if not OnTarget(v, t_v) then {
12              if a_i ≠ '#' then w_i := Evaluate(v, −∞, ∞);
13              w_v := UpdateValue(v);
14              return ;
15          }
16      } /* for ... */
17  } else /* Cutnode(v, t_v) */{
18      for i := 1 to b do {
19          if v_i ∉ Γ'(v) then Γ'(v) := Γ'(v) ∪ {v_i};
20          if ExaminationUseful(v_i, t_v) then {
21              if v_i ist noch nie bewertet worden then {
                    if v ist MIN-Knoten then w_i := Evaluate(v_i, β_v, β_v + 1);
                    else w_i := Evaluate(v_i, β_v − 1, β_v);
22              } else w_i := Evaluate(v_i, −∞, ∞); }
23          w_v := UpdateValue(v);
24          if (die Anzahl der Nachfolger v_k von v mit OnTarget(v_k, t_v) und nicht
25              ExaminationUseful(v_k, t_v)) > γ_v then return
26      } /* for ... */
27      if not OnTarget(v, t_v) then
28          for all v_i ∈ Γ'(v)[= Γ(v)] do {
29              if a_i ≠ '#' then w_i := Evaluate(v_i, −∞, ∞);
30              w_i := UpdateValue(v_i);
31          } }
```

Abb. 2.14: Teil-Expansion

Bemerkung: Man beachte noch einmal, daß wir eine Unterscheidung zwischen billigen Schrankenbewertungen und teuren Punktbewertungen machen. Wenn ein Knoten zum ersten Mal bewertet wird, sind die Bewertungskosten niedrig. Die vorzugebende Schranke für diese Bewertung läßt sich vom Target des Vaterknotens herleiten. Wenn ein Knoten vorher schon einmal bewertet wurde, wird er mit einer teuren Bewertung noch einmal bewertet. Knoten, die einen Punktwert besitzen, werden nie wieder bewertet.

```
bool ExaminationUseful(node v = (a_v, z_v), target t = ((α, β), δ, γ))
    if v ist noch nie bewertet worden then return true;
    else {
        if a_v ∈ {'≥' , '#' } and z_v ≥ β and α = '≥' then return false;
        if a_v ∈ {'≤' , '#' } and z_v ≤ β and α = '≤' then return false;
        /* v kann sicher ein Target bekommen. */
        if not OnTarget(v, t) then return false;
        /* v scheint nicht für ein Teiltarget von t geeignet. */
        return true; /* Man kann nicht sagen, ob v ein Kandidat für ein Target ist*/
    }
```

Abb. 2.15: Soll man v bewerten?

Die Prozedur ExaminationUseful(...) informiert PartialExpansion darüber, ob ein Knoten bewertet werden sollte oder ob schon genügend Informationen über den Knoten vorliegen. PartialExpansion(...) braucht nur diejenigen Knoten zu bewerten, bei denen der Wert weder dem Target des Vorgängers widerspricht noch ihn unterstützt. Die Nachfolger mit solch ungeklärtem Zustand aufzuspüren, ist die Aufgabe von der Prozedur ExaminationUseful(...).

```
value Evaluate(node v, int α*, int β*)
    z_v = EvaluatePosition(v, α*, β*)
        /* Führe eine konventionelle Bewertung aus, */
    if z_v ≥ β* then a_v := '≥' ;
    else if z_v ≤ α* then a_v := '≤' ;
    else a_v := '#' ;
    return (a_v, z_v);
        /* und transferiere das Ergebnis in einen Wert w_v = (a_v, z_v). */
```

Abb. 2.16: Bewertung eines Knotens

Die Bewertungsprozedur Evaluate(...) nutzt zunächst eine konventionelle Bewertungsfunktion, die eine Zahl z liefert. Ein Fenster [α^*,β^*] bestimmt, innerhalb welchen Inter-

valls der genaue Zahlwert von v interessant ist. Abhängig von α^*,β^* und z wird ein Wert gebildet.

```
bool OnTarget(node v, target t_v)
    /* let t_v = ((α_v, β_v), δ_v, γ_v) sei das Target und
         w_v = (a_v, z_v) der Wert des Knotens v. */
    if α_v = '≥' and β_v > z_v and a_v ∈ {'#', '≤'}
        return false;
    else if α_v = '≤' and β_v < z_v and a_v ∈ {'#', '≥'}
        return false;
    else return true;
```

Abb. 2.17: OnTarget

Wenn die Prozedur Cc2s(. . .) beginnt, einen Knoten v zu untersuchen, ist immer ein Target t_v mit diesem Knoten assoziiert. Ein Target stellt eine Anforderung an einen Knoten dar, die etwas mit dem Wert des Knotens v zu tun hat. Wenn z.B. der Wert von v $('\#', x)$ ist und das Target einen Wert $\leq x$ fordert, passen Anforderung und Wirklichkeit zusammen. Wir gehen dann optimistisch davon aus, daß eine Erweiterung des Suchbaums unterhalb von v die Sicherheit erhöhen wird, mit der der Wert von $v \leq x$ ist. Wir gehen außerdem davon aus, daß es sinnvoll ist, unter v weitere Knoten zu erzeugen, und daß deshalb unser Algorithmus auf dem richtigen Wege (engl. on target) ist, wenn er v untersucht.

Wenn aber der Wert von v in diesem Beispiel $('\geq', x+1)$ ist oder wird, sehen wir wenig Chancen, daß das Target t_v erfüllt werden kann. In diesem Fall steht nämlich der Wert von v im Widerspruch zu der Wertkomponente von t_v, und Cc2s orientiert sich um.

2.1.5 Komplexes Beispiel

Ausgangspunkt der folgenden Beispiele ist wiederum die Spitze des Spielbaums aus Abbildung 2.2.

Erste Hypothese

Wir starten damit, daß uns der Algorithmus eine Entscheidung liefern soll, die auf Strategien beruht, deren Blätter mindestens in Tiefe drei liegen (Conspiracy Number $C = 1$). In der DetermineMove Prozedur werden die Knoten $v.1$ und $v.2$ erzeugt und mit einem Fenster $(-\infty, \infty)$ bewertet. Daraufhin hält der Algorithmus den Zug $(v, v.1)$ für den besten, da $w_{v.1} = (\text{'\#'}, 5)$ und $w_{v.1} = (\text{'\#'}, 4)$ ist. (Genau, wie in Abschnitt 2.1.3).

Aufbau von Hülle und Strategien unter $v.1$

In der Prozedur DetermineMove wird $v.1$ das Target $((\text{'}\geq\text{'}, 5), 2, 1)$ und $v.2$ das Target $((\text{'}\leq\text{'}, 5), 2, 1)$ zugewiesen. Cc2s(...) wird mit dem Knoten $v.1$ aufgerufen. Dort wird erstmal nach geeigneten Nachfolgern Ausschau gehalten. In PartialExpansion(...) werden die Knoten $v.1.1$ und $v.1.2$ erzeugt und bewertet. Beide Nachfolger bekommen den Wert $w_{v.1.1} = w_{v.1.2} = (\text{'}\geq\text{'}, 5)$. Die Knoten $v.1.1$ und $v.1.2$ bekommen die Targets $((\text{'}\geq\text{'}, 5), 1, 1)$. Es folgt ein Aufruf von Cc2s(...) mit dem Knoten $v.1.1$. Mit Hilfe von PartialExpansion(...) wird der Knoten $v.1.1.1$ aufgedeckt. Er erhält ebenfalls den Wert $(\text{'}\geq\text{'}, 5)$. Nach dem Cc2s(...)-Aufruf mit diesem Knoten, der sofort mit OK zurückkehrt, kehrt Cc2s(...) auch beim Knoten $v.1.1$ mit OK zu seinem Vorgänger zurück. Die Knoten $v.1.2$, $v.1.2.1$ und $1.2.2$ werden analog den Knoten $v.1.1$, $v.1.1.1$ und $1.1.2$ erzeugt und bewertet. Damit ergibt sich folgender Suchbaum, nachdem Cc2s(...) auch für den Knoten $v.1$ beendet worden ist:

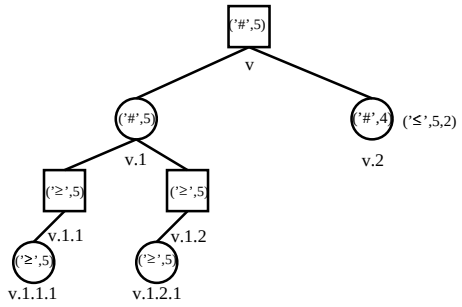


Abb. 2.18: Schnappschuß vom Suchbaum (1)

Fehlhypothesen beim Aufbau von Hülle und Strategien unter $v.2$

Der linke Teilbaum ist abgearbeitet, Knoten $v.2$ muß noch das Target $((\text{'}\leq\text{'}, 5), 2, 1)$ erfüllen. Cc2s(...) wird nun also mit dem Knoten $v.2$ und dem Target $((\text{'}\leq\text{'}, 5), 2, 1)$ aufgerufen. $v.2$ ist ein CUT-Knoten, und es genügt, einen Nachfolger zu finden, dessen Minimaxwert kleiner oder gleich 5 ist. Deshalb wird zunächst der Knoten $v.2.1$ erzeugt und bewertet. Er erhält den Wert $(\text{'}\leq\text{'}, 5)$. Der Algorithmus verzweigt zu $v.2.1$. Von dort aus wird der Knoten $v.2.1.1$ erzeugt und bewertet. Da dieser einen Wert größer oder gleich 5 anzeigt, wird $v.2.1.1$ in PartialExpansion(...) sofort nochmal genauer bewertet. Der Knoten erhält den Wert $(\text{'\#'}, 8)$. Dann wird am Knoten $v.2.1$ UpdateValue(...) aufgerufen. $v.2.1$ hat den Wert $(\text{'}\leq\text{'}, 5)$, aber ein Sohn hat einen größeren Wert. Der Wert $w_{v.2.1}$ wird zu $(\text{'\#'}, 8)$ gesetzt. Der Knoten $v.2.1$ kann sein Target scheinbar nicht mehr erfüllen, deshalb kehrt Cc2s(...) mit NOT_OK zu $v.2$ zurück. An $v.2$ wird der nächste Sohn voruntersucht. Er sieht ebenfalls vielversprechend aus. $v.2.2$ bekommt ein Target $((\text{'}\leq\text{'}, 5), 1, 1)$, und Cc2s(...) verzweigt zu $v.2.2$. Auch hier wird zunächst der Sohn $v.2.2.1$ erzeugt und bewertet. Er wird im zweiten Anlauf mit $(\text{'\#'}, 7)$ bewertet. Cc2s(...) ruft UpdateValue(...) mit dem Knoten $v.2.2$ auf, und $v.2.2$ bekommt den Wert $(\text{'\#'}, 7)$. Cc2s(...) springt zu $v.2$ und stellt fest, daß auch Knoten $v.2.2$ mit NOT_OK zurückgekehrt ist und daß $v.2$ kein vielversprechender Kandidat mehr ist, sein Target zu erfüllen. UpdateValue(...) liefert für den Knoten $v.2$ den Wert $(\text{'\#'}, 7)$, im Anschluß bekommt auch die Wurzel den Wert $(\text{'\#'}, 7)$. Die Prozedur DetermineMove(...) konnte also den Zug $(v, v.1)$ nicht als den besten nachweisen, sondern es sieht so aus, als sei der andere Zug der bessere. Es stellt sich folgende Situation dar:

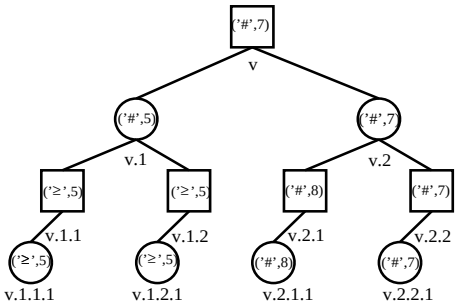


Abb. 2.19: Schnappschuß vom Suchbaum (2)

Neue Orientierung

Die Prozedur DetermineMove(...) gibt nun an $v.2$ das Target $((\text{'}\geq\text{'}, 7), 2, 1)$ und an $v.1$ das Target $((\text{'}\leq\text{'}, 7), 2, 1)$. $v.2$ wird zuerst bearbeitet. Im folgenden bekommen $v2.1$,

$v2.2$, $v2.1.1$ und $v2.2.1$ die Targets $(('\geq' , 7), 1, 1)$, $(('\geq' , 7), 1, 1)$, $(('\geq' , 7), 0, 0)$ und $(('\geq' , 7), 0, 0)$, die alle erfüllt werden. Cc2s(...) bearbeite also wieder den Knoten $v.1$, dem immer noch das Target $t_{v.1} = (('\leq' , 7), 2, 1)$ zugeordnet ist. Es werden offenbar Nachfolger von $v.1$ benötigt, die helfen können, $(('\leq' , 7), 1, 1)$ zu erfüllen. Es bieten sich $v.1.1$ und $v.1.2$ an. Von beiden weiß man nur, daß es Teilbäume unter ihnen gibt mit Minimaxwerten größer oder gleich 5. PartialExpansion(...) wird bemüht, $v.1.1$ nochmals genauer zu bewerten. Da $v.1.1$ schon mal bewertet wurde, wird eine teure Bewertung benutzt, die sicherstellt, daß $v_{v.1.1}$ das Attribut '#' bekommt und danach nie wieder bewertet werden wird. Nach einem weiteren Aufruf von UpdateValue(...) für $v.1.1$ hat dieser Knoten den Wert $('\#' , 6)$. Damit wurde ein Sohn gefunden, der geeignet erscheint, das Target $t_{v.1.1} = (('\leq' , 7), 1, 1)$ zu erfüllen. Cc2s(...) springt zu $v.1.1$. Beide Nachfolger müssen einen Zahlwert ≤ 7 haben, damit $t_{v.1.1}$ erfüllt wird. PartialExpansion(...) sorgt im Zusammenspiel mit Evaluate(...) dafür, daß $v.1.1.1$ den Wert $('\#' , 5)$ und $v.1.1.2$ den Wert $('\leq' , 7)$ bekommen. Cc2s(...) wandert noch einmal zur Wurzel hoch, immer mit der Rückgabe OK. Die Routine DetermineMove(...) bekommt bestätigt, daß es unter $v.1$ eine Strategie gibt, deren Blätter in Tiefe 3 liegen, und deren Minimaxwert ≤ 7 ist. Damit ist der Zug $(v, v.2)$ der Beste. Der endgültige Suchbaum sieht folgendermaßen aus:

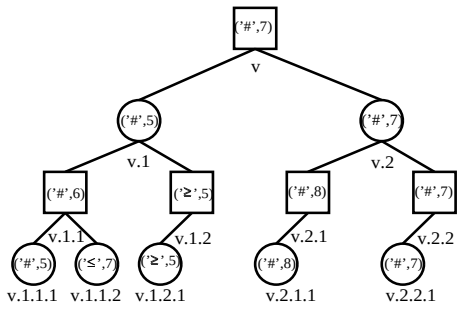


Abb. 2.20: Endgültiger Suchbaum

Blattdisjunkte Strategien

Bisher haben wir unser neues Verfahren nur auf Hüllen mit konstanter Tiefe t angewendet. Wir möchten nun aber auch nicht rein tiefenorientierte Strategien betrachten, sondern wir fordern zusätzlich eine weitere, die Entscheidung unterstützende Strategie, die im Suchbaum enthalten sein sollen.

Wir starten bei der Wurzel des Suchbaums von Abbildung 2.18. $v.1$ bekommt das Target $t_{v.1} = (('\leq' , 7), 2, 2)$, $v.2$ bekommt $t_{v.2} = (('\geq' , 7), 2, 2)$. Da $v.2$ den höheren Wert hat, wird dort zuerst gearbeitet. Die Prozedur Cc2s(...) läuft zu $v.2.1$, und dort stellt

die Prozedur PartialExpansion(...) fest, daß ein weiterer Nachfolger von $v.2.1$ mit einem Wert ≥ 7 nützlich wäre, um das Target $t_{v.2.1} = (('\leq' , 7), 1, 2)$ zu erfüllen. Eine Bewertung von $v.2.1.2$ liefert das gewünschte Ergebnis. Ebenso wird an $v2.2$ verfahren und die Cc2s(...) Routine läuft bis zur Wurzel zurück. Sie wird nun auf $v.1$ angesetzt und läuft bis zum Knoten $v1.1.1$ herunter, der dabei das Target $t_{v.1.1.1} = (('\geq' , 7), 0, 2)$ zugewiesen bekommt. Die Mindestsuchtiefe ist also erreicht, aber es müssen zwei blattdisjunkte Strategien gefunden werden, die den Wert $('\geq' , 7)$ am Knoten $v1.1.1$ belegen. Deshalb werden $v.1.1.1.1$ und $v.1.1.1.2$ erzeugt und bewertet. An $v.1.1.2$ wird analog wie bei Knoten $v.1.1.1$ verfahren, so daß wir folgenden besuchten Spielbaum erhalten:

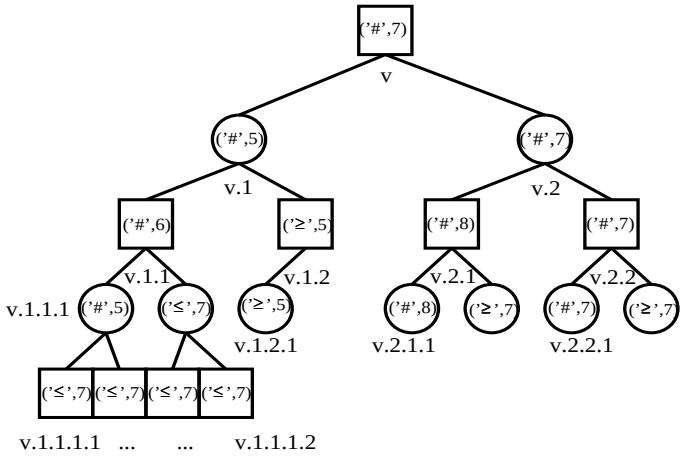


Abb. 2.21: Zwei blattdisjunkte Entscheidungsstrategien

2.2 Hergeleitete Eigenschaften

2.2.1 Vergleich zwischen der Cc1s und dem $\alpha\beta$-Algorithmus

Wir versuchen nun, unseren Algorithmus mit dem bislang erfolgreichsten Spielbaum Suchalgorithmus, dem $\alpha\beta$-Algorithmus zu vergleichen. Weil der nicht mit Sicherheitsaspekten wie Conspiracy Numbers umgehen kann, beschränken wir uns bei der Analyse auf Hüllen, deren Blätter alle die gleiche Entfernung zur Wurzel aufweisen (sog. fixed-depth Hüllen). Wir verzichten auf die Forderung, mehrere blattdisjunkte Beweisstrategien herauszuarbeiten, und nennen diese Variante 'Controlled Conspiracy Number 1 Search' (Cc1s).

Ferner ist die Analyse des $\alpha\beta$-Algorithmus für gewöhnlich auf fixed-depth full-width Spielbäume mit einem uniformen Verzweigungsgrad beschränkt. Wir übernehmen auch diese Einschränkung.

Sei nun G ein fixed-depth full-width Spielbaum mit Tiefe d und Breite b. Um den besten Zug zu bestimmen, berechnet der $\alpha\beta$-Algorithmus den Minimaxwert der Wurzel von G. Dazu muß er mindestens $b^{\lfloor \frac{d}{2} \rfloor} + b^{\lceil \frac{d}{2} \rceil} - 1$ Blätter bewerten [KM75].

Satz 2.2-3 **(Aufwand)**
Der Cc1s-Algorithmus muß mindestens $b^{\lceil (d-1)/2 \rceil} + (b-1) \cdot b^{\lfloor (d-1)/2 \rfloor}$ Blätter von G bewerten, um den besten Zug zu finden. Im besten Fall wird diese Schranke auch erreicht. Bezüglich der Anzahl der zu bewertenden Blätter ist dies optimal. □

Satz 2.2-4 **(Terminierung)**
Wenn G endlich ist (insbesondere, wenn G ein fixed-depth full-width Spielbaum ist), terminiert der Cc1s-Algorithmus in endlicher Zeit. □

Wegen der Schleife in Zeile 3 in Abb. 2.7 ist der Nachweis der Terminierung nicht trivial.

Satz 2.2-5 **(Korrektheit)**
Der Entscheidungszug, den die Cc1s findet, basiert auf dem Minimaxwert von G, d.h., jeder minimax-basierte Suchalgorithmus kommt in folgendem Sinne zu dem gleichen Ergebnis: Sei $m = (v, v')$ der vom Cc1s-Algorithmus ermittelte beste Zug, $m' = (v, v'')$ ein von einem anderen minimax-basierten Verfahren ermittelter bester Zug. Dann sind die Minimaxwerte von v' und v'' gleich. Wenn der beste Zug eindeutig ist, sind m und m' gleich. □

Die entsprechenden Beweise befinden sich am Ende dieses Kapitels.

2.2.2 Cc2s im Vergleich zu Auswahl-Expansion-Aktualisierung basierten Algorithmen

Satz 2.2-6 **(Partielle Korrektheit)**
Wenn der Algorithmus seine Arbeit beendet hat, basiert das Ergebnis auf einer Hülle, die nach dem Minimax-Prinzip ausgewertet wurde. □

Sofern sie die gleiche Hülle auswerten, ist damit sichergestellt, daß unser Algorithmus und ein Algorithmus, der für jede Spielbaumerweiterung die drei Phasen Auswahl, Expansion und Aktualisierung durchführt, zu dem gleichen Ergebnis kommen (vgl. Abschnitt 1.5).
Unsere neue Technik hat zwei Vorteile gegenüber Algorithmen, die ein Blatt expandieren,

indem alle Nachfolger angehängt werden. Deren neu erzeugte Nachfolger wissen nämlich nicht, zu welchem Zweck sie erzeugt wurden. In der Praxis ist es oft wesentlich schneller zu entscheiden, ob der heuristische Wert eines Knotens größer oder kleiner einer vorgegebenen Zahl ist, als einen punktgenauen Wert zu bestimmen. Darüber hinaus brauchen wir nicht immer alle Nachfolger eines Knotens zu erzeugen.

Die gesteigerte Effizienz unseres Verfahrens hängt zum einen vom Verzweigungsgrad des Spiels, zum anderen davon ab, wievielmal schneller die Bewertungsprozedur eine Aussage über eine Wertschranke machen kann, als den heuristischen Punktwert zu liefern.

Es überrascht nun nicht mehr, daß es kein konkurrenzfähiges Schachprogramm gibt, das auf der Auswahl-Expansions-Aktualisierungs Technik basiert.

Satz 2.2-7 **(Terminierung und Fortschritt beim Cc2s-Algorithmus)**
In unendlich großen Spielbäumen kann es passieren, daß der Cc2s-Algorithmus nicht terminiert, da es natürlich beliebig große Spielbäume gibt, die nicht die erforderlichen blattdisjunkten Strategien enthalten. In diesem Fall ist aber Fortschritt in dem Sinne gewährleistet, daß der Suchbaum wächst. Der Algorithmus kann also nicht in eine Endlosschleife geraten. Die Argumentation ist analog zu der von Satz 2.2-4.

In einem beliebigen, endlich großen Spielbaum G terminiert der Cc2s-Algorithmus zwar immer (analog zu Satz 2.2-4), es kann aber passieren, daß das Ergebnis rein formal nicht den vorgegebenen Sicherheitsanforderungen (bzgl. Entfernung der Blätter zur Wurzel und bzgl. der Conspiracy Number) entspricht. Da das aber nur dadurch passieren kann, daß Teilergebnisse auf Werten von nichtexpandierbaren Blättern von G beruhen, bei denen man die spieltheoretischen Werte kennt, nehmen wir das gerne in Kauf. □

2.3 Heuristiken zur Beschleunigung

Der Vollständigkeit halber stellen wir nun noch einige Heuristiken zur Beschleunigung und Verbesserung des Verfahrens vor, wie sie im Schachprogramm ConNerS eingesetzt werden. Es handelt sich um Heuristiken, die entweder aus Veröffentlichungen über Varianten des $\alpha\beta$-Algorithmus übernommen wurden, oder um Heuristiken, die durch Probieren und einfache Programmoptimierung auf kleinen Benchmarks zustande gekommen sind. Sie fallen somit aus dem Rahmen dieser Arbeit heraus. Trotzdem sind sie für den Erfolg unseres Schachprogramms P.ConNerS unverzichtbar.

Mehrmaliges Bewerten von Knoten

Im vorigen Kapitel war es eine wichtige Eigenschaft der Prozedur PartialExpansion, daß sie dafür sorgt, daß jeder Knoten höchstens zweimal bewertet wird. Dadurch war eine Terminierungsargumentation für den Gesamtalgorithmus erst möglich geworden.

Sei nun v ein Knoten, dessen heuristischer Wert bisher $('\geq' , z_v)$ war. Wenn diese Information für vernünftigen Fortschritt der Cc2s-Prozedur nicht mehr ausreicht, wird v neu bewertet. Anstatt den Wert aber sofort in einen '#' -Wert umzuwandeln, ist es auch möglich, v erstmal im Fenster $(z_v, z_v + \delta)$ ($\delta \in \mathbb{Z}$) neu zu bewerten. Liefert die Bewertungsfunktion eine Zahl z zwischen z_v und $z_v + \delta$, ist der neue Wert von v gleich $('\#' , y)$ mit $y \in [z_v, z_v + \delta]$; liefert sie eine Zahl größer als $z_v + \delta$, wird der Wert zu $('\geq' , z_v + \delta)$. Die Terminierung des Gesamtalgorithmus ist natürlich nach wie vor gesichert, allerdings verschlechtert sich das Worstcase-Verhalten.
Für unser Schachprogramm ConNerS hat sich $\delta = 80$ als gut erwiesen. 80 Punkte entsprechen 0.8 Bauerneinheiten.

Vorsortierung

Wenn ein Blatt v des Suchbaums expandiert werden muß und das zu v gehörige Target t_v v zu einem Cut-Knoten macht, genügt es oft, nur ein oder zwei Nachfolger v's zu bewerten. Hierbei ist eine heuristische Vorsortierung der neu generierten Züge nützlich. Da sich die vom Cc2s-Verfahren erzeugten Knoten nahe der Suchbaumwurzel befinden, ist die Geschwindigkeit, mit der solch eine Vorsortierung geschieht, nur minder relevant. Alle neu erzeugten Züge werden mit Hilfe einer kleinen $\alpha\beta$-Ruhesuche vorbewertet und sortiert.

Fenster an der Wurzel

Bisher bestimmt die Prozedur DetermineMove zu Beginn einer Iteration denjenigen Zug mit dem höchsten z_1-Wert als den mutmaßlich auch in Zukunft besten Zug und erzeugt ein Target $t_1 := ('\geq' , z_1, d-1, C)$ für den besten Nachfolger und $t_i := ('\leq' , z_1, d-1, C)$ für die anderen Nachfolger der Spielbaumwurzel. Es hat sich für den praktischen Gebrauch als sehr nützlich erwiesen, diese Targets aufzuweichen, so daß (für ein $\delta \in \mathbb{Z}$) $t_1 := ('\geq' , z_1 - \delta, d-1, C)$ und $t_i := ('\leq' , z_1 + \delta, d-1, C)$ für $i > 1$ gesetzt wird. Dadurch wird ein Zug auch dann als bester akzeptiert, wenn sein Wert während der Berechnung ein wenig absinkt oder der Wert eines anderen Nachfolgers ein wenig ansteigt. Kleine Schwankungen kann man i.allg. vernachlässigen. Im Schachprogramm ConNerS hat sich $\delta = 7$, was sieben hundertstel Bauerneinheiten entspricht, als vernünftiger Wert

herausgestellt.

Die Einführung von Hauptvarianten

Wir haben im Schachprogramm ConNerS die Möglichkeit vorgesehen, nicht nur nach einer bestmöglichen Entscheidung zu suchen, sondern auch eine Hauptvariante zu ermitteln. Obwohl das ein wenig zusätzliche Rechenzeit kostet und auch kein Spielstärkezuwachs beobachtet werden konnte, bieten sich drei Vorteile:

- Der wichtigste Vorteil besteht darin, daß man nicht nur einen guten Zug an der Wurzel erhält, sondern auch einen Folgezug für den Gegner, den man dann als Ratezug einsetzen kann. Auf diese Weise läßt sich auch die Zeit, in der der Gegner am Zug ist, sinnvoll nutzen.
- Mit Hilfe der Hauptvarianten erhält man als Beobachter eines Spiels bessere Einblicke in die internen Rechenvorgänge. Das Aufspüren von Fehlern ist auf diese Weise wesentlich einfacher.
- Manchmal konnten wir beobachten, daß das Schachprogramm ConNerS zwar die richtigen Züge vorschlug, aber mit einer aus schachlicher Sicht falschen Begründung. Das führte in Einzelfällen dazu, daß das Programm viel Rechenzeit in Teilbäume investierte, die anders besser hätte eingesetzt werden können.

Wir erreichen das Herausbilden von Hauptvarianten dadurch, daß wir erweiterte Targets einführen, sog. '#' -Targets. Sie haben die Form

$$t = (\text{'\#'}\ , z, \delta, \gamma, z', \delta', \gamma')).$$

Wie normale Targets auch formulieren sie eine Anforderung an den Knoten, nur daß sie in der Lage sind, von einem Knoten zu fordern, daß sein Minimaxwert $x \geq z'$ und gleichzeitig $\leq z$ ist. Dabei soll die obere Schranke sicher sein mit Conspiracy Number γ und Resttiefe δ. Die untere Schranke soll auf die gleiche Weise mit γ' und Resttiefe δ' gesichert werden.

Transpositionstabellen

Eine Transpositionstabelle ist eine Hashtabelle für Knoten. Sie dient dazu, Teilergebnisse, die in einem Teilbaum errechnet wurden, in einem anderen Teilbaum nicht neu berechnen zu müssen. Ein Eintrag in eine Transpositionstabelle besteht aus einem Tupel (s, t). Dabei ist die Komponente s ein 32-Bit Schlüssel, der zur Identifikation eines Eintrags dient. Die Tatsache, daß der Schlüssel nicht eindeutig ist, wird wie z.B. beim Computerschach

üblich, ignoriert. Die Komponente t enthält dasjenige Target, das zuletzt am Knoten v, dessen Hashschlüssel s ist, erfüllt wurde. Der Effekt, den eine Transpositionstabelle beim Cc2s-Verfahren hat, ist bei weitem nicht so groß wie z.B. beim $\alpha\beta$-Verfahren. Bei letzterem kann die Tabelle auch zur Zugsortierung herangezogen werden. Da beim Cc2s-Verfahren der Suchbaum komplett vorliegt, ist die Transpositionstabelle für die Zugsortierung unerheblich.

Die Hashfunktion wurde folgendermaßen konstruiert:

Für jede mögliche Figur auf jedem möglichen Feld wurden zwei 32-Bit Zufallszahlen ermittelt. Die XOR-Operation über alle auf dem Brett befindlichen Figuren liefert dann zwei verschiedene 32-Bit Schlüssel. Der eine wird zur Identifikation in den Hashtabelleneintrag mit hineingeschrieben, der andere liefert durch Modulobildung einen Index in die Tabelle.

Wir benutzen drei verschiedene Transpositionstabellen: Je eine für '#' -Targets, für '$\leq$' -Targets und für '$\geq$' -Targets. Die Tabellen für '$\leq$' - und '$\geq$' -Targets enthalten 10000 Elemente, die für '#' -Targets enthält 5000 Elemente. Die Operationen 'read' und 'write' und 'remove' sehen jeweils folgendermaßen aus:

Wenn an einem Knoten v ein Target t_v erfüllt werden konnte, wird dies in die zu t_v gehörige Transpostionstabelle eingetragen.

void write(Index ix, Eintrag (Schlüssel s, Target $t = ((\alpha, \beta), \delta, \gamma)$))

```
1  /* Seien s der Hashschlüssel und T^{q∈{≤,≥,#}} die Transpositionstabellen für
2  die entsprechenden Targets, T_i^q sei der i-te Eintrag von T^q. */
3  /* ' <' sei für Targets t = ((α,β),δ,γ) und t' = ((α',β'),δ',γ') so definiert:
4       t < t' ⇔ (γ < γ' or (γ = γ' and δ < δ')) */
5  if (t ist ein ≤-Target oder ein #-Target /* also α ∈ {'≤' , '#' } */)
6       if (T_ix^≤ ist leer oder t ≥ T_ix^≤.t) /* Targetkomponente */ then
7            T_ix^≤ := (s,t);
8  if (t ist ein ≥-Target oder ein #-Target)
9       if (T_ix^≥ ist leer oder t ≥ T_ix^≥.t) then  T_ix^≥ := (s,t);
10 if (t ist ein #-Target)
11      if (T_ix^# ist leer oder t ≥ T_ix^#.t) then  T_ix^# := (s,t);
```

Abb. 2.22: Schreibe Hash-Tabelleneintrag

Bevor die Suche unter einem inneren Knoten v gestartet wird, sieht der Suchalgorithmus in den Transpositionstabellen nach, ob das Target t_v, oder ein höherwertiges schon einmal an anderer Stelle erfüllt werden konnte. Wenn das der Fall ist, wird v sofort wieder

boolean read(Index ix, Anfrage (Schlüssel s, Target $t = ((\alpha, \beta), \delta, \gamma)$))
$/*$ Seien s der Hash-Schlüssel und $T^{q \in \{'\leq', '\geq', '\#'\}}$ die Transpositionstabellen für die entsprechenden Targets, T_i^q sei der i-te Eintrag von T^q. $*/$
if (t ist ein $\leq$-Target) {
  if ($T_{ix}^{\leq}.s = s$) **and** (t ist schon erfüllt, wegen $T_{ix}^{\leq}.t$) **then** return true;
} else **if** (t ist ein $\geq$-Target) {
  if ($T_{ix}^{\geq}.s = s$) **and** (t ist schon erfüllt, wegen $T_{ix}^{\geq}.t$) **then** return true;
} else **if** (t ist ein $\#$-Target) {
  if ($T_{ix}^{\#}.s = s$) **and** (t ist schon erfüllt, wegen $T_{ix}^{\#}.t$) **then** return true;
}
return false;

Abb. 2.23: Lese Hash-Tabelleneintrag

verlassen, mit der Meldung an den Vorgänger von v, daß das Target t_v erfüllt werden konnte.

Wenn ein Target t_v an einem Knoten v nicht erfüllt werden konnte, ist es nicht klar, was sinnvollerweise in die Transpositionstabelle eingetragen werden kann.

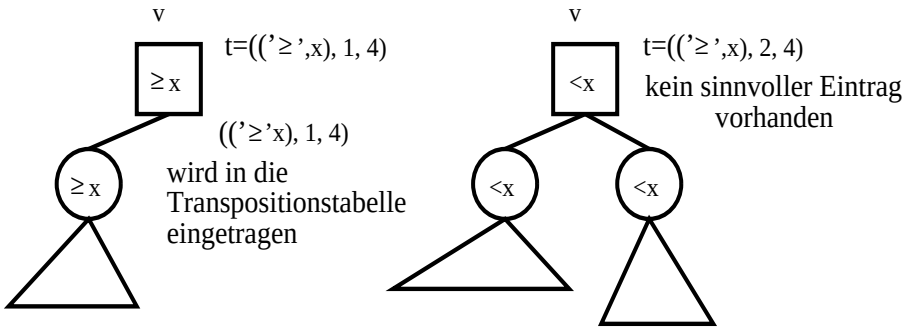


Abb. 2.24: Verwertbarkeit von fehlgeschlagenen Suchen

Sehen wir uns dazu das Beispiel aus Abbildung 2.24 an. Dort wurde am Knoten v zunächst einmal das Target $t = (('\geq', x), 1, 4)$ erfüllt. In einer späteren Phase des Suchens sollte ein Target $(('\geq', x), 2, 4)$ erfüllt werden. Das ist jedoch fehlgeschlagen, und es wurde vom Suchalgorithmus ein neuer Wert für v hochgereicht, der kleiner als der alte Wert ist. Nachdem die Suche fehlgeschlagen ist, haben wir mehrere Möglichkeiten: Z.B. können wir den alten Eintrag in der Transpositionstabelle stehen lassen. Das ist allerdings riskant, da ja gerade zuvor gezeigt wurde, daß sich das in der Transpositonstabelle stehende Er-

gebnis nicht halten läßt. Mit welcher Sicherheit man die neuen Erkenntnisse abspeichern sollte, ist aber auch nicht zu sehen. Deshalb löschen wir in solchen Fällen den entsprechenden Eintrag in der Transpositionstabelle.

void remove(Index ix, Eintrag (Schlüssel s, Target $t = ((\alpha, \beta), \delta, \gamma)$))
  if (t ist ein $\leq$-Target)
    if ($T^{\leq}_{ix}.s = s$) **then** lösche $T^{\leq}_{ix}$;
  if (t ist ein '$\geq$' -Target)
    if ($T^{\geq}_{ix}.s = s$) **then** lösche $T^{\geq}_{ix}$;
  if (t ist ein '#' -Target)
    if ($T^{\#}_{ix}.s = s$) **then** lösche $T^{\#}_{ix}$;

Abb. 2.25: Lösche Hash-Tabelleneintrag

Stabilität von Bewertungen

Damit der Cc2s-Algorithmus zielgerichtet arbeiten kann, müssen die Bewertungen an den Blättern neben einer gewissen Qualität auch Stabilität besitzen. Da die Suche hauptsächlich durch die Bewertungen gesteuert wird, führt eine instabile Bewertung zu Terminierungsschwierigkeiten. Für das Schachprogramm P.ConNerS haben sich Tiefe-2 $\alpha\beta$-Suche einschließlich Ruhesuchen, Verkürzungen und den üblichen Spielbaumerweiterungstechniken als gut erwiesen. Reine Ruhesuchen führen die Suche zu oft in eine falsche Richtung, während Tiefe-3 $\alpha\beta$-Suchen zu teuer sind.

Aufteilung von Targets erst in der Nähe von Blättern

Für unser Schachprogramm hat es sich als besonders wirkungsvoll erwiesen, das Aufteilen der Conspiracy-Zahlen auf mehrere Nachfolger erst in der Nähe von Blättern zu gestatten. Aufgeteilt wird demnach ein Target erst, wenn der Resttiefenanteil des Targets kleiner als 2 ist.

Nullmoves und Fail-High Reduktionen

Nullmoves und Fail-High Reduktionen werden, wie in der Literatur [Don93] [Fel96] für $\alpha\beta$-Suchen veröffentlicht, eingesetzt.

Sicherheit von Zügen an der Wurzel

Sei z_ϵ der Zahlwert der Wurzel und Paare (a, b) seien Anforderungstupel, die besagen, daß der Zug an der Wurzel mit Tiefe a und Conspiracy Number b sicher sein soll. ConNerS

versucht, im Mittelspiel die Züge mit der Anforderungsfolge (1,1), (2,1), (3,1), (4,1), (4,2), (5,2), (. . . ,2) sicher zu machen und in elementaren Endspielen (Endspiele, bei denen nur noch Bauern und maximal je ein Läufer oder ein Springer auf dem Brett sind) mit der Anforderungsfolge (1,1), (2,1), (3,1), (4,1), (4,2), (4,3), (5,3), (6,3), (7,3), (8,3), (8,4), (8,5), (9,5), (. . . ,5).

2.4 Experimentelle Ergebnisse

Wir beginnen mit einer Leistungsbeurteilung, die zum einen aus der Auswertung einer Serie von Teststellungen besteht (dem BT2630 Test) und zum anderen aus einer Serie von Spielen gegen Cheiron'97. Die übrigen Abschnitte beschäftigen sich mit technischen Fragen. Wir zeigen, daß der Geschwindigkeitsgewinn unseres Target-gesteuerten Verfahrens gegenüber Verfahren, die bei jeder Expansion alle Nachfolger mit Punktwerten auswerten, praxisrelevant ist. (Zur Erinnerung: einen Wert der Form $w = (\text{'\#'}, z)$ nennen wir auch Punktwert; $w = (\text{'}\leq\text{'}, z)$ oder $w = (\text{'}\geq\text{'}, z)$ nennen wir auch Schrankenwert;) Außerdem beschäftigen wir uns mit der Frage, inwieweit sich die bei ConNerS entstehenden Suchbäume von Suchbäumen mit 'fester Suchtiefe' (aber durchaus unter Einfluß von Nullmoves und Fail-High Reduktionen) unterscheiden.

Der BT2630 Test

Der erste Teil besteht aus Stellungen des sogenannten BT2630 Tests [BT94]. Er besteht aus 30 Stellungen (vgl. auch Anhang A). Jede ist mit einer großmeistergerechten Lösung versehen. Ein Programm bekommt exakt 15 Minuten Zeit, um einen Zug auszurechnen. Für Stellungen, die nicht gelöst werden, gibt es 900 Strafpunkte. Für solche, die gelöst werden, gibt es die Anzahl der Sekunden bis zur Lösung als Strafpunkte. Findet und verliert ein Programm den richtigen Lösungszug während der erlaubten 15 Minuten mehrfach, zählt der letzte Zeitpunkt, an dem die Lösung entdeckt und bis zum Abbruch beibehalten wurde.

Der Test weist dem Ergebnis des Rechners eine Pseudo-ELO-Zahl zu. (Das ELO System ist ein statistisches Maß, um relative Spielstärke von Schachspielern zu messen. Es hat sich zur Bewertung der Spielstärke von Schachspielern durchgesetzt.) Den BT2630 Test betreffend hält ConNerS dem Vergleich mit Topprogrammen wie Fritz und Hiarcs stand. Die Meßergebnisse der fremden Programme wurden von uns auf uns zur Verfügung stehender Hardware ermittelt. Man muß hier fairerweise hinzufügen, daß eine Einprozessorversion von ConNerS im Spiel keineswegs mit Programmen wie Fritz oder Hiarcs mithalten kann. Dafür ist auch der investierte Zeitaufwand zur Erstellung und Feinoptimierung der Programme nicht vergleichbar.

Ergebnis auf dem BT2630-Test

Programm auf Maschine	Pseudo-ELO
ConNerS auf Sparc 144 MHz	2375
ConNerS auf Sparc 300 MHz	2408
ConNerS auf Pentium II 450 MHz	2444
Cc1s-Suche auf Pentium II 450 MHz	2403
Cheiron'97 auf Sparc 144 MHz	2331
Fritz 4.0 auf 200Mhz Pentium	2373
Hiarcs 6.0 auf 200Mhz Pentium	2403

Abb. 2.26: Testergebnis

Obwohl der Schwerpunkt des Tests auf taktischer Spielstärke liegt, ist entscheidend, daß ConNerS die taktischen Lösungen findet, obwohl es eine fein-granulare Bewertungsfunktion besitzt. Hier hebt sich das Cc2s-Verfahren deutlich von den Ergebnissen der konventionellen CNS ab [Sch90].

Testspiele

Der zweite Teil des Testens beruht auf Spielen. Wir wollten uns der Herausforderung stellen, unser auf Cc2s beruhendes Schachprogramm mit Programmen zu vergleichen, die den $\alpha\beta$-Algorithmus benutzen einschließlich der allgemein anerkannten Verbesserungen: Transpositiontables, diverse Sortierheuristiken [Sch83], Fail High Reduktionen, Nullmoves u.s.w.

Wie schon bei anderen Gelegenheiten [Fel96] [ADLR93] haben wir 25 Startstellungen vorgegeben. Wir ließen ConNerS zwei Spielserien von je 50 Spielen gegen das Programm Cheiron'97 machen. Cheiron'97 benutzt den Negascout-Algorithmus [Rei89], Fail High Reduktionen, Transpositiontables, Killerheuristiken, Rückschlag- und Schacherweiterungen. Das Programm und seine Vorgänger haben gute Ergebnisse bei Computerschachturnieren erzielt. Der Vorgänger Ulysses wurde z.B. Neunter bei der Computerschachweltmeisterschaft 1992. Der Hauptvorteil, Cheiron'97 als Gegner einzusetzen war jedoch, daß wir eine Tiefe-2 Suche des Programms als Bewertung von ConNerS nehmen konnten. Damit war es möglich, einen Vergleich der Suchalgorithmen zu erreichen.

Die erste Serie fand auf einer Sparc 144 Mhz Maschine statt, und die Programme bekamen 8 Stunden Zeit für je 40 Züge. Der Vergleich endete 26.5 zu 23.5 zugunsten von Cheiron'97. Obwohl sich die Einzelergebnisse einer zweiten Serie, bei der die Programme 4 Stunden für 40 Züge Zeit hatten (Turnierzeitmodus), unterscheiden, endete auch diese

Serie mit 26.5 zu 23.5. Beide Serien zusammen führen also zu einem Ergebnis von 53 zu 47. Wir interpretieren dieses Ergebnis so, daß Cheiron'97 und ConNerS ungefähr gleich stark sind, auch wenn diese 100 Spiele keine statistisch signifikante Aussage zulassen.

2.4.1 Baumbeschneidungen und schnelle Bewertungen

Nr.	Zeit(sec) (V_1)	#Knoten	Zeit (V_2)	#Knoten	Zeit (V_3)	#Knoten
1	36	4613	115	4322	889	32382
2	7	5695	29	6548	203	47837
5	58	30230	424	29628	420	35403
6	40	8658	200	7223	892	33992
7	20	7576	110	5106	446	25875
8	12	4498	42	4245	231	20064
9	30	13288	128	11439	669	63248
11	12	2461	38	2474	324	16433
12	67	7134	220	4763	652	15851
13	14	3165	75	2614	391	11362
14	25	6690	70	6301	430	41579
15	23	6496	124	6177	573	30218
17	9	2380	17	2249	163	11555
18	72	38632	188	16844	660	53347
19	80	13400	121	7573	859	49707
20	3	1542	12	1464	127	12124
21	138	4294	182	3677	628	14391
22	3	972	13	945	107	6817
23	69	5256	258	5119	733	14839
24	11	2490	100	2645	301	9626
25	48	11979	101	7065	635	43446
26	6	3657	29	3052	371	26285
27	15	3343	36	2300	858	64954
29	17	2771	87	3001	551	16824
30	132	23790	194	11318	585	38188
$\sum$	947	215010	2913	158092	12698	736347

Abb. 2.27: Nutzen der Targets

Es wurde bereits mehrfach erwähnt, daß wir der Überzeugung sind, einer der Hauptvor-

teile der mit Hilfe von Targets top-down kontrollierten Suche gegenüber herkömmlicher Auswahl-Expansions-Aktualisierungs-Suche sei, daß man mit Hilfe der Targets nicht immer alle Nachfolger eines Knotens bewerten muß und außerdem noch schnelle Bewertungen einsetzen kann, die statt Punktwerten lediglich Schranken zurückliefern.

Um den Nutzen der Targets für unser Schachprogramm sichtbar zu machen, haben wir drei verschiedene Versionen von ConNerS erzeugt und auf dem BT2630 Test einander gegenübergestellt. Version V_1 entspricht der normalen Version von ConNerS. V_2 bewertet jeden Knoten, der bewertet werden soll, sofort mit einem heuristischen Punktwert (d.h., jeder Knoten wird mit einer Tiefe-2 $\alpha\beta$-Suche mit dem Startfenster $[-\infty, \infty]$ bewertet), und V_3 erzeugt bei jedem Expansionsschritt alle Nachfolgeknoten und weist ihnen einen heuristischen Punktwert zu.

Auf jeder Teststellung mußte jede der drei Versionen nach einer Entscheidung mit vorgegebener Sicherheit suchen. Für jede einzelne Stellung war die geforderte Sicherheit für alle Versionen gleich. Die jeweils verbrauchte Zeit und die jeweilige Anzahl der Suchknoten (ohne diejenigen Knoten, die von der Bewertungsprozedur, die aus einer Tiefe-2 $\alpha\beta$-Suche besteht, erzeugt werden) wurden dann miteinander verglichen. Die Stellungen 3,4,10, 16 und 28 wurden aus dem Test herausgenommen, weil V_3 zu langsam ist, um dort vernünftig meßbare Vergleichsergebnisse zu liefern.

In der Summe (s. Abb. 2.27) benötigte V_3 mit 12698 Sekunden 13,4 mal so lange wie V_1. Mit 2913 Sekunden brauchte V_2 immerhin 3,1 mal solange wie V_1. Zusätzlich erwähnt sei, daß die Suchergebnisse nur in den Stellungen 2, 25, 27 und 29 voneinander abweichen. Innerhalb dieser kleinen Untermenge von Stellungen hat lediglich V_1 in Stellung 25 die richtige Lösung gefunden. Man kann also davon ausgehen, daß die Entscheidungsqualität der drei Versionen ähnlich ist, wenn sie nach Entscheidungen mit derselben Sicherheit suchen.

Interessant ist, daß V_2 weniger Knoten (74%) als V_1 erzeugt hat. Das liegt sicher daran, daß V_2 dadurch daß diese Version immer mit Punktwerten arbeitet, ab und an einen genaueren Eindruck der Situation besitzt als V_1. Die Punktwerte sind genauer und erlauben eine bessere heuristische Steuerung der Suche.

2.4.2 Form der Suchbäume

Weiterhin stellt sich die Frage, wie die Suchbäume aussehen, die vom Programm ConNerS durchsucht werden. Allgemein ist das schwer zu beantworten, da sich die Suche den Gegebenheiten des Spiels anpaßt. Einen Eindruck sollen aber die folgenden Überlegungen geben:

Wir betrachten dazu die Anzahl der untersuchten Knoten pro Ebene nach einer längeren Berechnungszeit. Bei einem Tiefe-t $\alpha\beta$-Suchalgorithmus dürfte man einen exponentiellen Anstieg der Knoten mit einem Faktor sechs pro Ebene erwarten. Bei einer Tiefe-t Suche mit Nullmoves und Fail-High-Reduktionen sieht die Sache schon anders aus. Abb. 2.28 zeigt die jeweilige Knotenanzahl einer mit unserem Verfahren durchgeführten Tiefe-t Suche. Wir hatten diese Version auch schon als Cc1s-Suche bezeichnet. Allerdings haben wir hier alle in Abschnitt 2.3 vorgestellten Verbesserungen mit einbezogen. Gezählt wurden nur die Knoten, die vom Cc1s-Algorithmus erzeugt wurden. Die Knoten der Tiefe-2 $\alpha\beta$-Suchen und der dazugehörigen Ruhesuchen sind darin nicht enthalten. Gezeigt werden die Stellungen 1,5,9,13,17,21 und 25 des BT2630 Tests (Abb. 2.28). Der Meßzeitpunkt ist jeweils derjenige, an dem zum letzten Mal eine Iteration beendet wurde, bei einer maximalen Laufzeit von 15 Minuten.

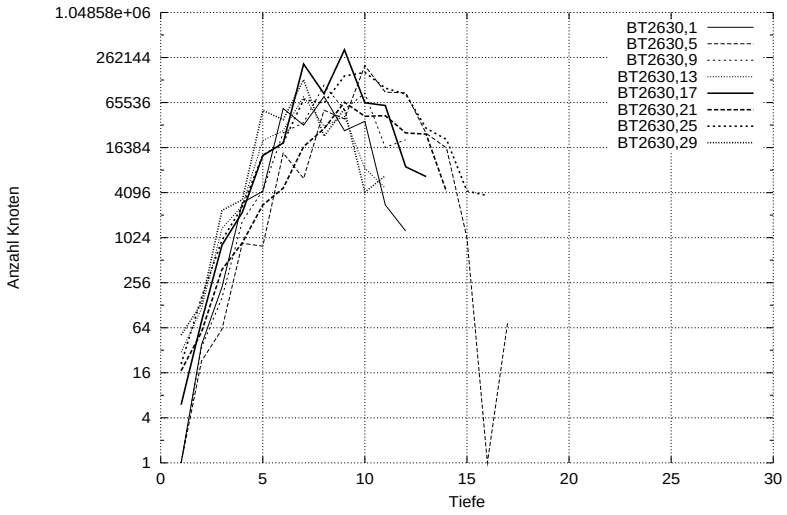


Abb. 2.28: Anzahl der Knoten pro Ebene des Cc1s-Verfahrens

Man sieht deutlich die Wirkung der vorgenommenen Variantenverkürzungen. Das Wachstum der Knotenzahlen über die Ebenen 1 bis 15 ist nicht exponentiell. Zum Ende werden die Knoten pro Ebene sogar wieder weniger (Ebenen 10-15). Trotzdem kann sich ein Verfahren, das auf Verkürzungen beruht, dem exponentiellen Wachstum der Strategien in der Nähe der Wurzel nicht entziehen (Ebenen 1-8), da dort immer alle Nachfolger betrachtet werden.

Bei Cc2s-Suchen werden zumindest einige Varianten wesentlich tiefer untersucht als bei der Cc1s-Variante (teilweise mehr als 10 Halbzüge). Das Diagramm 2.29 vermittelt einen visuellen Eindruck davon, wie die Zahl der Knoten über die verschiedenen Ebenen von Cc2s-Suchbäumen verteilt ist.

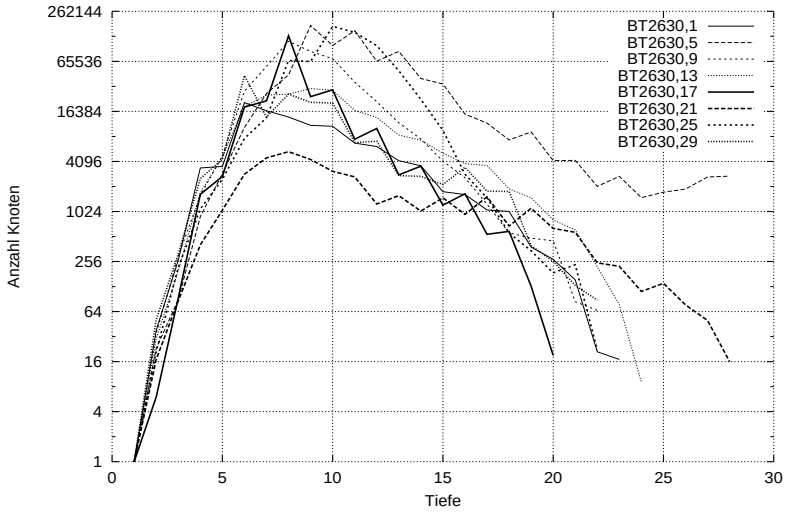


Abb. 2.29: Anzahl von Knoten pro Ebene des Cc2s-Verfahrens

Da die Anzahl der Knoten pro Ebene mit fortschreitender Tiefe stark sinkt, kann es sich nur um wenige, vereinzelte Varianten handeln, die der Cc2s-Algorithmus in den Tiefen 15 bis 27 verbringt. Auf einem Pentium 450 Mhz ist die Performance der Cc2s-Variante trotzdem um 41 Pseudo-Elo Punkte besser, als die der Cc1s-Variante. Es werden also Varianten verlängert, die (zumindest) das BT2630 Testergebnis merklich verbessern.

2.5 Beweise zu Kap. 2

Satz 2.1-2: Sei $G = ((V, K), h)$ ein Spielbaum, und sei $T(t)$ ein Suchbaum, also ein Teilbaum von G, der von einem Suchalgorithmus $\mathcal{A}$ zu einem Zeitpunkt t untersucht wurde. G und $T(t)$ sollen die gleiche Wurzel haben. Bezeichne außerdem $T_v(t)$ den Teilbaum von $T(t)$, der v als Wurzel besitzt.

Evaluate(...) sei die oben gegebene Prozedur, die Knoten statische Werte zuordnet. Solche statischen Werte brauchen zunächst nicht dem Minimax-Prinzip zu gehorchen. Evaluate(...) kann gefragt werden, wie sich der heuristische statische Zahlwert x eines Knotens v zu einer vorgegebenen Zahl $a \in \mathbb{Z}$ verhält. Falls $x = a$ ist, wird v der Wert $(\text{'\#'}\ , a)$ zugewiesen. Falls $x < a$ ist, wird v $(\text{'}\leq\text{'}\ , a - 1)$ zugewiesen, und wenn $x > a$ ist, bekommt v den Wert $(\text{'}\geq\text{'}\ , a + 1)$. Darüber hinaus kann Evaluate(...) auch gezwungen werden, einen 'punktgenauen', heuristischen Wert von v (also $h(v)$) zurückzuliefern. Der Wert von v ist dann $(\text{'\#'}\ , x)$, für einen Zahlwert x.

$\mathcal{A}$ soll UpdateValue(...) und Evaluate(...) in folgender Weise gebrauchen: Wenn $\mathcal{A}$ den Spielbaum erweitert, bewertet er das neue Blatt mit Hilfe von Evaluate(...). Zu jedem Zeitpunkt t' ist es $\mathcal{A}$ darüberhinaus erlaubt, Werte von Evaluate(...) beliebigen Knoten zuzuweisen. Wann immer $\mathcal{A}$ zur Zeit t' Evaluate(...) aufruft, wendet er den Operator UpdateValue(...) auf alle Knoten des Pfades von v zur Wurzel von $T(t')$ (das ist auch die Wurzel von G) an. Er geht dabei bottom-up vor.

Wenn $\mathcal{A}$ die Wurzel von $T(t)$ erreicht, gelten folgende Eigenschaften für alle Knoten von $T(t)$.

/1 $w_v = (\text{'}\leq\text{'}\ , x)$ impliziert, daß es eine Hülle unterhalb von v gibt, deren Minimaxwert $\leq x$ ist.

/2 $w_v = (\text{'}\geq\text{'}\ , x)$ impliziert, daß es eine Hülle unterhalb von v gibt, deren Minimaxwert $\geq x$.

/3 $w_v = (\text{'\#'}\ , x)$ impliziert, daß es eine Hülle unterhalb von v gibt, deren Minimaxwert $\leq x$ ist, und daß es eine möglicherweise andere Hülle unterhalb von v gibt, deren Minimaxwert $\geq x$ ist. Die beiden Hüllen brauchen nicht dieselben zu sein.

/4 Sei w_t die Wertkomponente eines Targets t, die von w_v (einem Wert eines Knotens v) unterstützt wird. Wenn v durch t zu einem ALL-Knoten wird, widerspricht kein Wert der Nachfolger von v dem Wert w_t. Wenn v durch t zu einem CUT-Knoten wird, gibt es einen Nachfolger von v, dessen Wert dem Wert w_t nicht widerspricht.

Beweis: Wir untersuchen die Entstehung eines Wertes w_v rückwärts in der Zeit. Dafür betrachten wir einen Knoten v zu einem Zeitpunkt t. Wir betrachten weiterhin alle Möglichkeiten, wie dieser Wert von v zustande gekommen sein kann, wie also w_v vor dem letzten

Aufruf von UpdateValue(...) beim Knoten v ausgesehen haben kann. Wir nehmen dabei an, daß Satz 2.1-2/1 bis 2.1-2/3 vor dem letzten Aufruf des UpdateValue(...) Operators für den Knoten v und den unter ihm hängenden Teilbaum gültig war.

Es ist offensichtlich, daß Satz 2.1-2 für Blätter von $T(t)$ immer gilt.

Sei nun t der gegenwärtige Zeitpunkt, kurz nachdem UpdateValue(...) am Knoten v beendet wurde. Sei t^* der Zeitpunkt genau vor dem Aufruf von UpdateValue(...). Nehmen wir an, daß Satz 2.1-2 zur Zeit t für alle Knoten unterhalb von v gültig ist und daß es für alle Knoten gültig ist, die nicht auf dem Pfad von v zur Wurzel liegen. Außerdem nehmen wir an, daß Satz 2.1-2/1,2.1-2/2 und 2.1-2/3 zum Zeitpunkt t^* auch für den Knoten v gültig war.

Im folgenden beziehen sich Zeilennummern auf den Updateoperator der Abbildung 2.8 auf Seite 34.

$t^* \to t$: Sei oBdA. v ein MAX-Knoten. Sei w_v^t der Wert von v zum Zeitpunkt t.

1. $w_v^t = (\text{'}\geq\text{'}, x)$ Wenn v ein neues Blatt ist, ist Satz 2.1-2/2 wahr, wegen der Definition eines Wertes und wegen des Verhaltens von Evaluate(...). Ansonsten war das Attribut a_v des Wertes von v auch '$\geq$' zum Zeitpunkt t^* (weil sich Attribute nur von '$\leq$' nach '#' oder von '$\geq$' nach '#' ändern können). Wegen Zeile 8 ist w_v^t durch Zeile 6 entstanden, da sonst a_v^t '#' wäre. Weil in Zeile 6 der alte Zahlwert eines Knotens in den neuen mit einfließt, kann x durch die letzte Ausführung von UpdateValue(...) nicht abgesenkt worden sein. Offensichtlich war zum Zeitpunkt t^* der Wert $(\text{'}\geq\text{'}, y)$ mit einem $y \leq x$.

   (a) Einer der Nachfolger hat den Wert $(\text{'}\geq\text{'}, x)$ und bestimmt z_v^t. Weil wir annehmen, daß Satz 2.1-2/2 zur Zeit t^* für alle Knoten unterhalb von v wahr war, folgt, daß Satz 2.1-2/2 auch zum Zeitpunkt t wahr ist.

   (b) $w_v^{t^*}$ bestimmt w_v^t: Falls der Zahlwert $z_v^{t^*}$ des Knotens v $(\text{'}\geq\text{'}, x)$ war, wählen wir für Satz 2.1-2/2 denselben Teilbaum mit Wurzel v aus, wie schon zum Zeitpunkt t^*. Man beachte: Wenn v gerade zuvor bewertet wurde und die aktuelle UpdateValue-Phase verursacht hat, besteht der gesuchte Teilbaum nur aus dem Knoten v selber.

2. $w_v^t = (\text{'}\leq\text{'}, x)$

   Wenn v ein neues Blatt ist, ist Satz 2.1-2 wahr, wegen der Definition eines Wertes und wegen der Prozedur Evaluate(...). Sonst war das Attribut a_v zum Zeitpunkt t^* auch schon '$\leq$' (weil Attribute sich nur von '$\leq$' zu '#' oder von '$\geq$' zu '#' ändern können).

   (a) $\Gamma = \Gamma'$ und alle Nachfolger von v haben Werte mit Attributen '$\leq$' oder '#' und z_v^t = maximum$\{z_{v,i}^t, i \in \{1 \ldots b\}\}$: Für alle Teilbäume, deren Wurzeln

die Söhne von v sind, ist Satz 2.1-2/1 wahr. Wegen der Definition einer Beweisstrategie hält die Behauptung auch für v.

(b) $w_v^t = w_v^{t^*}$: Entweder wurde w_v^t durch eine direkte Bewertung bestimmt, so daß v selber verantwortlich für die aktuelle UpdateValue-Phase ist. Dann erfüllt der Spielbaum, der nur aus dem Knoten v besteht, den Satz. Ansonsten ist Satz 2.1-2/1 erfüllt, weil wir den gleichen Teilbaum wie zum Zeitpunkt t^* nehmen können.

3. $w_v^t = (\text{'\#'}\,, x)$

(a) $a_v^{t^*} = \text{'\#'}$:

i. $w_v^t = w_v^{t^*}$: Satz 2.1-2/3 ist mit derselben Argumentation wie zum Zeitpunkt t^* gültig.

ii. $z_v^t < z_v^{t^*}$: z_v^t könnte zum einen durch eine direkte Bewertung abgesenkt worden sein. Weil wir annehmen, daß $a_v^{t^*} = \text{'\#'}$ ist, liefert uns die Bewertung die Gültigkeit von Satz 2.1-2/3, nämlich durch den Spielbaum, der nur aus dem Knoten v und sonst nichts besteht. Zum anderen kann z_v^t nur noch durch Zeile 10 abgesenkt worden sein. Das bedeutet, daß $\Gamma = \Gamma'$ ist und alle Attribute der Nachfolger von v entweder '$\leq$' oder '#' sind und daß z_v das Maximum $z_{v.i}, i \in \{1 \ldots b\}$ der $v.1 \ldots v.b$ ist. Weil wir annehmen, daß Satz 2.1-2 für alle Knoten unterhalb von v gilt, gibt es eine Teilhülle unter v, deren Minimaxwert kleiner oder gleich z_v^t ist. Wir wissen aber auch, daß es eine Teilhülle geben muß, deren Minimaxwert größer oder gleich $z_v^{t^*}$ ist, und somit erst recht $\geq z_v^t$ ist, da $z_v^{t^*} \geq z_v^t$ ist. Satz 2.1-2/3 gilt also auch für diesen Fall.

iii. $z_v^t > z_v^{t^*}$: Die Teilhülle $T_v(t)$ des Suchbaums mit minimax(T_v) $\leq z_v$ ist dieselbe wie zum Zeitpunkt t^*. Wir bekommen den anderen Teilbaum, den wir noch brauchen, durch den Nachfolger, der den Wert von v dominiert, oder dadurch, daß v direkt bewertet worden war.

(b) $a_v^{t^*} = \text{'}\leq\text{'}$:

i. Zeile 9 verursachte, daß $a_v^t = \text{'\#'}$ ist: Wegen der Zeilen 2, 3 und 8 ist $\Gamma = \Gamma'$, und alle Nachfolger von v haben Attribute $\in \{\text{'}\leq\text{'}\,, \text{'\#'}\,\}$, und nur Nachfolger mit Attributen '#' haben Zahlwerte größer oder gleich $z_v^{t^*}$. Der gesuchte Teilbaum für Satz 2.1-2/1 zum Zeitpunkt t^* (bzw. falls eine Bewertung von v die aktuelle UpdateValue-Phase ausgelöst hat, die direkte Bewertung) liefert uns einen gewünschten Teilbaum. Derjenige Nachfolger, der z_v^t bestimmt, liefert den anderen Teilbaum.

ii. Zeile 5 hat verursacht, daß $a_v^t =$ '#' ist: Zeile 6 liefert einen Nachfolger von v, der z_v^t bestimmt. Durch diesen Nachfolger bekommen wir den einen Teilbaum. Weil es sogar einen Teilbaum unter v gibt, dessen Minimaxwert $\leq z_v^{t*}$ war und immer noch ist (so ein Teilbaum existiert ja weiter), gibt es einen gewünschten Teilbaum, dessen Minimaxwert $\leq z_v^t$ ist.

(c) $a_v^{t^*} =$ '$\geq$' : Dies impliziert, daß a_v^t durch Zeile 9 zu '#' gesetzt worden ist. Außerdem folgt daraus, daß $\Gamma = \Gamma'$ sein muß, daß es nur Nachfolger mit den Attributen'$\leq$' und '#' geben kann und daß alle Nachfolger von v, die Zahlwerte $\leq z_v^{t^*}$ haben, das Attribut '$\leq$' besitzen. Alle Nachfolger zusammen stellen sicher, daß ein Teilbaum T_v des Suchbaums existiert, dessen Minimaxwert $\leq z_v^t$ ist. Wir leiten den anderen Teilbaum vom Zeitpunkt t^* ab.

Zu Satz 2.1-2/4: Sei v ein beliebiger MAX-Knoten, w_v sein Wert. Sei w_t die Wertkomonente eines Targets t, die von w_v unterstützt wird.

Angenommen, der Knoten v werde durch t zu einem ALL-Knoten, und es gebe einen Nachfolger von v, dessen Wert w_t widerspricht. Dann hat w_t die Form $(\text{'}\leq\text{'}\ , x)$. Der Nachfolger, der w_t widerspricht hat die Form $(\text{'}\geq\text{'}\ , y)$ oder $(\text{'\#'}\ , y)$ mit $y > x$. In diesem Fall wäre aber w_v in den Zeilen 4-6 des Update-Operators (Abb. 2.8 auf Seite 34) auf $(\text{'\#'}\ , y)$ gesetzt worden. Damit würde w_v w_t nicht unterstützen. Das ist ein Widerspruch zur Annahme.

Nehmen wir nun an, v werde durch t zu einem CUT-Knoten, und die Werte aller Nachfolger von v widersprechen w_t. Dann hat w_t die Form $(\text{'}\geq\text{'}\ , x)$ und die Werte der Nachfolger von v haben die Form $(\text{'}\leq\text{'}\ , y)$, oder $(\text{'\#'}\ , y)$ mit $y < x$. Dann hätte aber UpdateValue den Wert für v mit Hilfe der Zeilen 8-10 (Abb. 2.8) ermittelt. Dann hätte w_v den Wert $(\text{'\#'}\ , z_v)$ oder $(\text{'}\leq\text{'}\ , z_v)$ mit $z_v \leq y$ bekommen. Dann widerspräche w_t aber w_v, was ein Widerspruch zu der Annahme ist, daß w_v w_t unterstützt. Für MIN-Knoten gilt alles analog.

Wenn die aktuelle UpdateValue-Phase die Wurzel von G erreicht, gilt Satz 2.1-2 für alle Knoten.

Satz 2.2-6: Wenn unser Algorithmus terminiert, basiert das Ergebnis an der Wurzel auf einer Hülle, die nach dem Minimax-Prinzip ausgewertet wurde. Ein beliebiger Minimax-Algorithmus käme zu dem gleichen Ergebnis, wenn er die gleiche Hülle untersuchte.

Beweis: Wir nehmen an, daß die Prozedur DetermineMove(. . .) beendet worden ist. Das bedeutet, daß alle Targets der Wurzelnachfolger erfüllt wurden. Die Targets der Nach-

folger der Nachfolger ebenfalls, u.s.w. Letztendlich gibt es eine Menge S von Knoten, die durch Zeile 1 der Prozedur Cc2s(...) erfüllt worden sind. Sei v ein Knoten aus S. Der Wert von v unterstützte offenbar die Wertkomponente des dazugehörigen Targets, als Cc2s(...) v zuletzt besucht hat. Der Wert von v ist entweder durch die Prozedur Evaluate(...) oder durch UpdateValue(...) entstanden. In beiden Fällen gibt es einen Teilbaum unter v, dessen Minimaxwert den Wert von v rechtfertigt. Das garantiert nämlich gerade Satz 2.1-2.

Satz 2.2-3: Wir nehmen an, daß der Spielbaum G, in dem wir einen besten Zug bestimmen wollen, uniform in Tiefe und Breite ist. Dies bedeutet, daß alle Blätter von G eine Entfernung d zur Wurzel haben und daß alle inneren Knoten genau b Nachfolger besitzen. Der Cc1s-Algorithmus muß mindestens $b^{\lceil (d-1)/2 \rceil} + (b-1) \cdot b^{\lfloor (d-1)/2 \rfloor}$ Blätter von G bewerten, um den besten Zug zu ermitteln. Was die Anzahl zu bewertender Blätter angeht, ist dies offensichtlich optimal.

Beweis: Der günstigste Fall tritt bei Cc1s(...) ein, wenn alle Teilhypothesen ohne Fehler verifiziert werden konnten. Das ist wegen der Konstruktion von Targets offensichtlich. In diesem Fall spannt der Cc1s-Algorithmus gerade eine MAX-Strategie der Tiefe $d-1$ für den besten Nachfolger auf und $b-1$ MIN-Strategien für die übrigen.

Satz 2.2-5 Der Entscheidungszug, den Cc1s(...) findet, basiert auf dem Minimaxwert von G. D.h., jeder Minimax-Algorithmus kommt zu demselben Ergebnis.

Beweis: Wegen der Art und Weise, wie Targets aufgeteilt werden, liegen alle Blätter der endgültigen Strategien mindestens in Ebene d. Weil G keine Knoten unterhalb der Tiefe d besitzt, befinden sich die Blätter der von Cc1s(...) erarbeiteten Strategien auch höchstens in Tiefe d. Satz 2.2-5 gilt dann wegen Satz 2.2-3.

Satz 2.2-4 Wenn G endlich ist (also insbesondere, wenn G ein Spielbaum mit fester Tiefe und konstantem Verzweigungsgrad ist), beendet Cc1s(...) seine Arbeit in endlicher Zeit.

Beweis: Nehmen wir an, DetermineMove(...) habe eine Verifikationsphase initiiert und Cc1s(...) gibt das Ergebnis NOT_OK zur Wurzel zurück. Wenn ein Knoten v zu seinem Vorgänger mit NOT_OK zurückkehrt, kann das nur zwei Gründe haben. Entweder wurde ein Sohn von v bewertet, oder einer der Nachfolger von v hat das Ergebnis NOT_OK zu v zurückgeliefert. Das kommt durch die Konstruktion von Targets zustande und dadurch, daß die Prozedur Split(...) nur gut geeigneten Knoten ein Target zuweist.

Weil jeder Knoten nur zweimal bewertet werden kann und weil es an den Blättern von G keine Nachfolger gibt, die ein NOT_OK zurückliefern können, terminiert DetermineMove(...) spätestens, wenn alle Knoten von G zweimal bewertet wurden. Für den Fall, daß G uniform ist mit Tiefe d und dem Verzweigungsgrad b, terminiert DetermineMove() nach höchstens $2 \cdot \sum_1^d b^t$ Bewertungen und einer anschließenden Verifikationsphase.

Kapitel 3

Der Parallele Cc2s-Algorithmus

Im folgenden werde ein *(Teil-)Problem* durch einen Tupel (v,t_v) repräsentiert , bei dem v ein Knoten des Spielbaums $G = ((V, E), h)$ und t_v ein zu v gehöriges Target ist. Gesucht wird dabei ein Teilbaum von G mit Wurzel v, der die Anforderungen erfüllt, die durch das Target t_v beschrieben werden.
Ein Aufruf der Prozedur Cc2s(v, t_v) liefert entweder einen solchen Teilbaum oder meldet, daß kein solcher Teilbaum gefunden wurde.

In diesem Kapitel beschreiben wir die Parallelisierung des Cc2s-Algorithmus. Dabei nutzen wir aus, daß wir, ohne gegenüber der sequentiellen Variante unnötige Mehrarbeit zu erzeugen, sehr viele Teilprobleme gleichzeitig bearbeiten könnten, wenn jedes gebildete Target erfüllt würde. Das entspricht zwar nicht der Realität, aber immerhin werden in der Praxis die meisten Targets erfüllt.

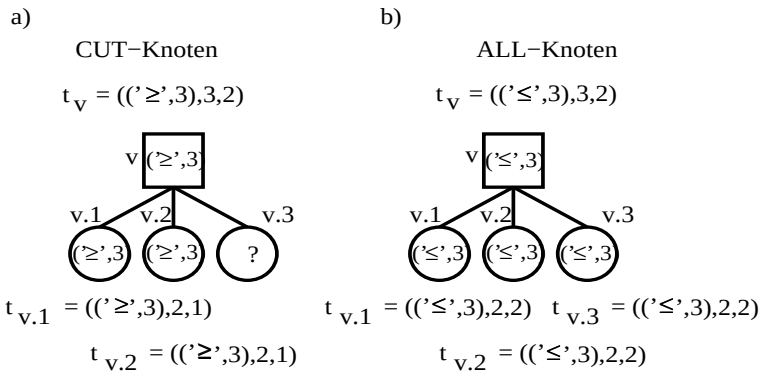


Abb. 3.1: Parallelität

Um in Abbildung 3.1b) das Target t_v zu erfüllen, müssen offenbar die Targets $t_{v.1} \ldots t_{v.3}$

alle erfüllt werden. Diese Targets $t_{v,1} \ldots t_{v,3}$ werden zur parallelen Verarbeitung freigegeben. CUT-Knoten bieten ebenfalls Parallelität, wenn zwei oder mehr Nachfolger Targets bekommen (Abbildung 3.1a)).

3.1 Hardware

Für die Experimente, die wir mit dem parallelen Cc2s-Verfahren in der Anwendung Schach gemacht haben, konnten wir verschiedene Rechnernetze und Parallelrechnerarchitekturen nutzen.

- Die Zukunft der Parallelrechner verschmilzt mit der der Workstationcluster. Die rasante Entwicklung der Standardprozessoren bei PCs und Workstations machen es erforderlich, auch beim Bau von Parallelrechnern auf Standardprozessoren wie z.B. Pentiumprozessoren zurückzugreifen.

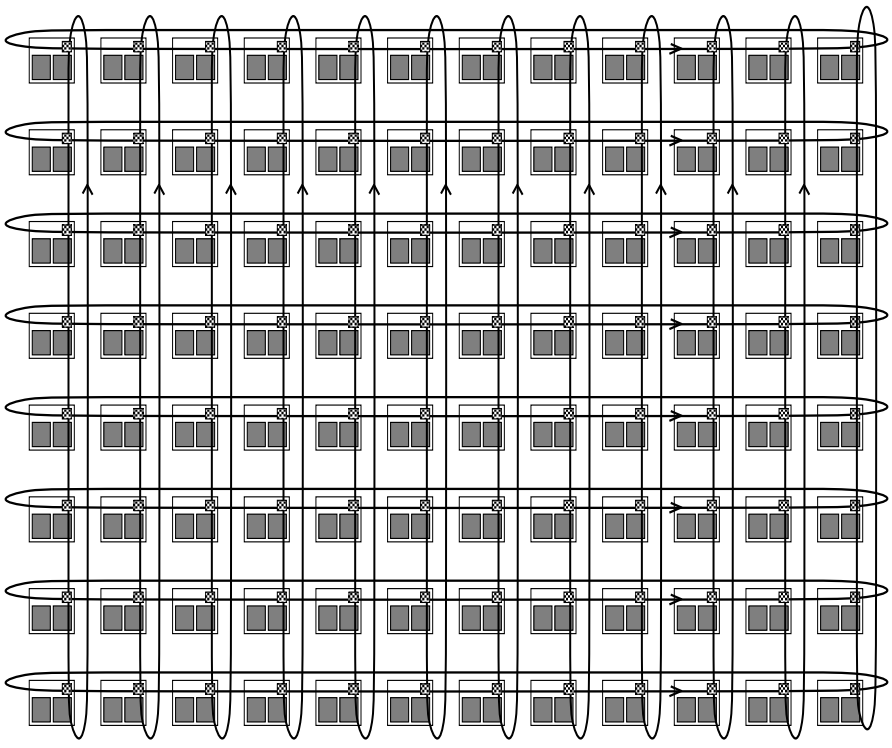


Abb. 3.2: Die Topologie des *hpcLine*-Rechners

Der *hpcLine*-Rechner der Universität Paderborn besteht aus 96 Siemens Primergy Rechnerknoten, die durch ein SCI-Netzwerk als 2D-Torus verbunden sind. Jeder einzelne Knoten besitzt 512 MByte Hauptspeicher, eine lokale 4 GByte Festplatte und zwei Intel Pentium II Prozessoren mit je 450MHz.

Mit Hilfe von 96 Scali/Dolphin PCI/SCI Interfacekarten mit 500 MByte/s SCI-Link Bandweite erreicht der Gesamtrechner 86.4 GFlop/s Peakperformance. SCI (Scalable Coherent Interface) ist ein IEEE Standard (IEEE Std. 1596-1992,[HH99]), der es ermöglichen soll, Rechnernetze mit sehr niedriger Latenz und sehr hohen Bandbreiten zu realisieren. Die Basiseinheiten eines SCI-Netzwerkes sind gerichtete Ringe (Abb. 3.3a)) und Rechenknoten. Ein Ring verbindet mehrere Rechenknoten durch unidirektionale Verbindungsstücke.

Mittlerweile gibt es auch für die SCI-Technologie Router (Abb. 3.3c)), die den Aufbau größerer Netze mit besseren Topologien ermöglichen. Der *hpcLine*-Cluster der Universität Paderborn besteht aus einem zweidimensionalen 12x8 Torus, bei dem die horizontalen und die vertikalen Verbindungen die erwähnten Ringe sind.

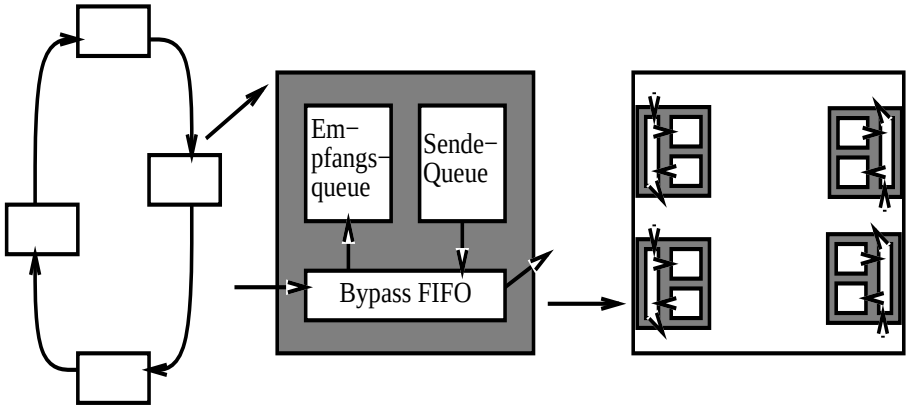


Abb. 3.3: SCI-Elemente

Eine Kommunikationstransaktion besteht aus zwei Teilen. Zum einen aus der Nachricht, die von einem Prozessor P_1 zu einem Prozessor P_2 geschickt werden soll, und aus einer Rückmeldung von Prozessor P_2 an P_1, daß die Nachricht auch wirklich angekommen ist. D.h., jede Nachricht muß einmal im Kreis wandern. Es gibt keine direkte Flußkontrolle zwischen benachbarten Prozessoren. Wollen mehrere Paare von Rechnern gleichzeitig kommunizieren, müssen sie sich die Bandbreite des

Rings teilen. Durch die Einfachheit des Ansatzes, die den Einsatz von sehr schneller Verbindungshardware und sehr einfachen Protokollen ermöglicht, können SCI Workstations im Rechnerverbund Leistungsdaten erreichen, wie sie bisher Parallelrechnern vorbehalten waren.

Wir haben für unsere Experimente fast ausschließlich diesen *hpcLine*-Workstationcluster verwendet.

- Der CC-48 Rechner besitzt 48 Rechenknoten, jeder davon ist mit einem PowerPC Prozessor MPC604 mit 133 MHz ausgestattet. Das Verbindungsnetzwerk besteht aus einem sogenannten 'fat mesh of clos', wie es in Abbildung 3.4 zu sehen ist. Die maximale Linkgeschwindigkeit beträgt 1Gbit/s. Auch bei diesem Rechner bildet jede Verbindung zwischen Prozessoren und Routern eine Punkt-zu-Punkt Verbindung.

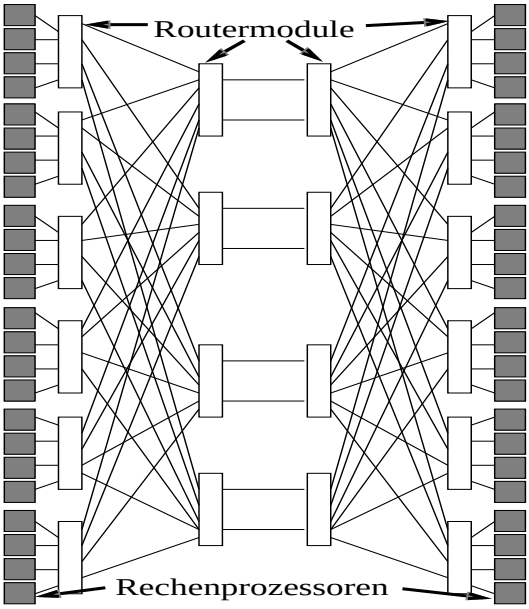


Abb. 3.4: Die Topologie des CC-48 Rechners

- Der GCel ist der älteste Rechner, den wir für Experimente benutzt haben. Es handelt sich dabei um 1024 T805 Prozessoren (30MHz), die als zweidimensionales Gitter angeordnet sind. Ein Gitter $G(n,m) = (P_G, L_G)$ ist definiert durch eine Menge P_G von Prozessoren und einer Menge L_G von Links (Leitungen), für die folgender Zusammenhang gilt:

$$P_G = \{(i,j) | 0 \leq i < n, 0 \leq j < m\}$$

und

$$L_G = \{\{(i,j),(i',j')\} | \; |i-i'| + |j-j'| = 1\}.$$

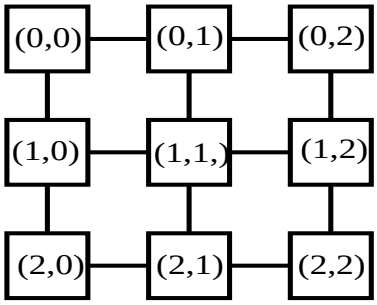


Abb. 3.5: Ein 3x3 - Gitter

Zwei Prozessoren, die durch einen Link verbunden sind, haben eine Punkt-zu-Punkt Verbindung zur Verfügung. Die Links ermöglichen eine Kommunikationsgeschwindigkeit von 20 MBit/s.

3.2 Bekannte parallele Spielbaumsuchverfahren

3.2.1 Schnelle Zugerzeugung und Bewertung

Bei Schachmaschinen wie BELLE [CT82] oder DEEP THOUGHT [Hsu90] (später DEEP BLUE) findet man feingranulare Parallelität, die auf Hardwareebene realisiert wird. Man kann verschiedene Kriterien einer Stellungsbewertung gleichzeitig ausführen oder mehrere Züge auf einmal erzeugen. Da diese beiden Aufgaben einen großen Anteil der Gesamtrechenzeit verbrauchen, wird eine erhebliche Beschleunigung des Schachprogramms erreicht. Das Programm CRAY BLITZ nutzte die Vektoreigenschaften des Cray-Prozessors zur schnellen Zugerzeugung und Bewertung aus [HGN85].

3.2.2 Parallele Fenstersuche

Baudet beschreibt eine Suche [Bau78], bei der verschiedene Prozessoren den gegebenen Spielbaum mit verschiedenen $\alpha\beta$-Startfenstern durchlaufen. Er ermittelte eine maximale Beschleunigung von 5 bis 6 auf zufällig generierten Bäumen. Wenn man bedenkt, wie nah z.B. beim Schach die tatsächlich durchsuchten Bäume an den sog. minimalen Bäumen (das sind die Teilbäume, die jedes Spielbaumsuchverfahren mindestens durchsuchen muß) liegen, wird klar, daß für Spielbäume von Anwendungen mit dieser Technik keine Beschleunigung erreicht werden kann:

Im bestmöglichen Fall besucht ein $\alpha\beta$-Algorithmus in einem Spielbaum mit konstantem Verzweigungsgrad b und Tiefe t

$$b^{\lceil t/2 \rceil} + b^{\lfloor t/2 \rfloor} - 1$$

Blätter des Spielbaums. Eine parallele Fenstersuche mit n Prozessoren und n verschiedenen Fenstern besucht mindestens

$$\underbrace{(n-1) \cdot b^{\lfloor t/2 \rfloor}}_{n-1 \text{ fehlschlagende Suchen}} + \underbrace{b^{\lceil t/2 \rceil} + b^{\lfloor t/2 \rfloor} - 1}_{\text{bestmöglicher Aufwand des } \alpha\beta\text{-Algorithmus}} \text{ Blätter.}$$

3.2.3 Spielbaumzerlegung

Alle erfolgreichen, vor dieser Arbeit bekannten parallelen Spielbaumsuchverfahren beruhen auf Varianten des $\alpha\beta$-Algorithmus und einer Zerlegung des Spielbaums.

Die ersten Versuche, den zu durchsuchenden Spielbaum zu zerlegen, bestanden darin, den Spielbaum in einen Prozessorbaum einzubetten. Mit 27 Prozessoren erreichten Finkel und Fishburn 1982 [FF82] einen Speedup von 5.31 in einem Prozessorbaum der Tiefe 3 und Breite 3.

1982 veröffentlichten Marsland und Campbell [MC82] den PV-Split-Algorithmus. Dabei werden rekursiv entlang der äußerst linken Variante erst alle Prozessoren im Teilbaum unter der linken Variante eingesetzt, bevor nach Abarbeitung dieses linken Teilbaums alle Prozessoren in allen rechten Teilbäumen gleichzeitig eingesetzt werden. Der PV-Split-Algorithmus fand breites Interesse in der Forschung [Sch89a] [HS89] und bildet die Basis für die meisten tatsächlichen Implementierungen paralleler Spielbaumsuchen. Das Fehlen dynamischer Lastverteilung führte dazu, daß die maximal erreichte Beschleunigung von 5 bis 6 nicht überschritten werden konnte. Newborn [New88] erreichte aber immerhin auf 8 Prozessoren eine Beschleunigung von 5.03 und Marsland, Olafsson und Schaeffer [MOS86] erreichten in ihrem Schachprogramm PHOENIX eine Beschleunigung von 3.75 auf 4 Prozessoren.

Den entscheidenden Durchbruch für die Spielbaumzerlegung brachte die dynamische Lastverteilung. 1988 stellen Ferguson und Korf [FK88] einen Algorithmus vor, der dynamische Lastverteilung ermöglicht, indem der Einsatz von Parallelität an allen Knoten erlaubt wird, an denen schon eine Grenze errechnet wurde. Sie haben das Verfahren in einem Dameprogramm implementiert.

Gleichzeitig wurde von Feldmann und anderen [FMMV90] das *Young-Brothers-Wait-Concept (YBWC)* entwickelt und in einem Schachprogramm ohne moderne Zugsortie-

rungsheuristiken eingesetzt. Beim YBWC wird an jedem Knoten der Einsatz von Parallelität so lange verboten, bis der erste Nachfolger fertig bearbeitet ist. Dieser Nachfolger liefert entweder einen Cutoff, oder man geht davon aus, daß es sich um einen Knoten handelt, an dem kein Cutoff stattfinden wird und erlaubt die parallele Bearbeitung der restlichen Nachfolger. Das eigentliche Loadbalancing wird durch einen Workstealing-Mechanismus erreicht.
In den folgenden Jahren wurde das YBWC weiterentwickelt und im Schachprogramm ZUGZWANG eingesetzt. Das Programm besaß eine sehr gute Zugsortierungsheuristik, und es wurden damit Speedups von 126 auf 256 Prozessoren und 344 auf 1024 Prozessoren erreicht [FMM91] [Fel93].

In der Zwischenzeit implementierten Otto und Felten [FO88] einen dynamischen Spielbaumsuchalgorithmus im Schachprogramm WAYCOOL. Ihr Algorithmus setzt Parallelität in Abhängigkeit von Einträgen der Transpositionstabellen ein. Parallelität darf an Knoten, für die es einen Transpositionstabelleneintrag gibt, erst eingesetzt werden, wenn die Suche unter dem Zug aus der Tabelle fertig ist. Im Prinzip handelt es sich hierbei um eine spezielle Variante des YBWC.

1990 stellte Hsu [Hsu90] einen zentralistischen Ansatz vor, den Spielbaum zu zerlegen. Ein sehr schneller Server bearbeitet einen Spielbaum bis zu einer recht geringen Suchtiefe t. Alle Knoten aus Ebene t werden aufgeteilt in Knoten des minimalen Baums (das sind die Knoten, die vom $\alpha\beta$-Algorithmus auf jeden Fall abgesucht werden müssen) und die restlichen. Daraus werden Prioritäten abgeleitet, in welcher Reihenfolge die Knoten zu untersuchen sind. Jeder Knoten wird dann von der Deep Blue Hardware mit einer weiteren Tiefe t' untersucht. In Simulationen gelang Hsu eine Beschleunigung von 350 auf 1000 Prozessoren. Leider ist nie öffentlich bekannt geworden, wie gut die tatsächlichen Speedups bei dem Schachprogramm Deep Blue gewesen sind. Immerhin muß man berücksichtigen, daß das Programm bereits 1997 den zu der Zeit als stärksten Spieler der Welt geltenden G. Kasparov in einem 6-Spiele Turnier besiegt hat.

Seit Anfang der neunziger Jahre wird am Massachusetts Institute of Technology (MIT) die auf ANSI-C basierende Programmiersprache CILK [BJK$^+$95] entwickelt. CILK wurde zu dem Zweck entwickelt, die Programmierung von verteilten Anwendungen zu erleichtern. Sie nutzt Parallelität auf der Ebene von Prozeduraufrufen und Threads, und die Lastverteilung wird durch Workstealing realisiert. Im Zuge der Bewertung von CILK wurden drei sehr gute Schachprogramme entwickelt: StarTech, $*$Sokrates und Cilkchess. Letzteres verwendet eine parallele Version des von A. Plaat vorgestellten MTD(f)-Algorithmus [Pla93]. Dieser wiederum verwendet bei der Suche Aufrufe eines $\alpha\beta$-Algorithmus mit

minimalem Startfenster. Die Parallelisierung des MTD(f)-Algorithmus reduziert sich somit auf die Parallelisierung eines $\alpha\beta$-Algorithmus.

3.3 Spielbaumzerlegung beim Cc2s-Algorithmus

3.3.1 Verwertung von Parallelität

Die im Prinzip zur Verfügung stehende Rechenleistung eines Parallelrechners oder eines Rechnerverbundes läßt sich nur in eine höhere Leistung und somit in eine verbesserte Entscheidungsqualität ummünzen, wenn es gelingt, die im Cc2s-Algorithmus vorhandene Parallelität sinnvoll zu nutzen.

Wenn beim Cc2s-Algorithmus das Target t_v eines Knotens v auf mehrere Nachfolger aufgeteilt wird, entstehen neue Teilprobleme, die im Prinzip gleichzeitig bearbeitet werden können. Diese neuen Teilprobleme werden in der parallelen Version des Cc2s-Algorithmus an *entfernte* Prozessoren gesendet und dort verarbeitet. (Ein Prozessor P_i ist entfernt von einem Prozessor P_j, wenn $i \neq j$ ist.)

Auf diese Weise bekommen wir einen parallelen Algorithmus, dessen Effizienz von den folgenden Faktoren abhängt:

1. Suchoverhead, der dadurch entsteht, daß Targets nicht erfüllt wurden:
   Das folgende Beispiel zeigt, daß eine parallele Bearbeitung von Teilproblemen dazu führen kann, daß der parallele Suchalgorithmus andere Ergebnisse als der sequentielle erzielt.

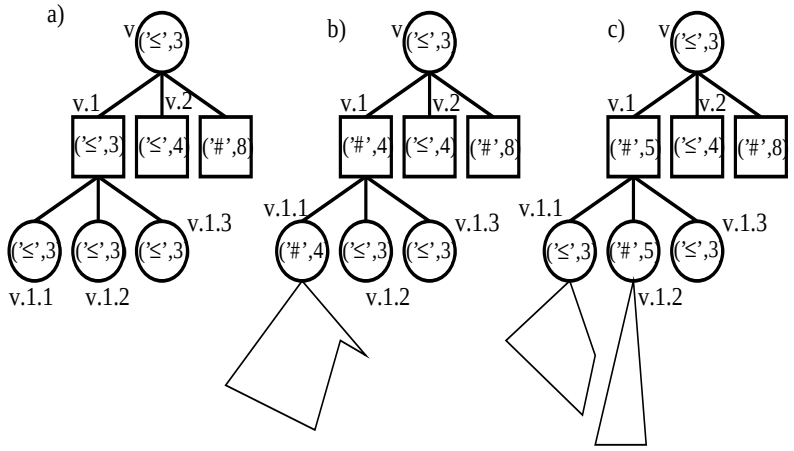


Abb. 3.6: Berechnungsausschnitt

Betrachten wir dazu folgendes Szenario: Sei v der Knoten in Abbildung 3.6a), der ein Target $t_v = (('\leq' , 3), \text{Resttiefe } 0, \text{Conspiracy Number } 2)$ bekommen hat. Der einzige Nachfolger von v, der dazu beitragen kann, daß t_v erfüllt wird, ist $v.1$. Dort werden die Targets $t_{v.1.1} = t_{v.1.2} = t_{v.1.3} = (('\leq' , 3), 0, 2)$ erzeugt. Der Algorithmus untersucht als erstes den Knoten $v.1.1$ und er weise aufgrund weiterer Berechnungen unter $v.1.1$ dem Knoten $v.1.1$ den Wert $('\#' , 4)$ zu. Es ist nun so, daß $t_{v.1.1}$ und damit t_v nicht erfüllt werden können. Außerdem bekommt $v.1$ den Wert $('\#' , 4)$ und v behält den Wert $('\leq' , 3)$. Wir haben die Situation von Abbildung 3.6b) erreicht. Der Cc2s-Algorithmus versucht nun, das Target t_v mit Hilfe des Knotens $v.2$ zu erfüllen. Wenn eine Bewertung von $v.2$ aber zu dem Wert $w_{v.2} = ('\#' , 4)$ führt, wird auch w_v zu $('\#' , 4)$. Es ist natürlich im folgenden Verlauf der Berechnungen möglich, daß der Knoten v das Target $t_v = (('\leq' , 4), 0, 2)$ zugewiesen bekommt. Die Knoten $v.1$ und $v.2$ sorgen dann dafür, daß t_v erfüllt wird.

Etwas ganz anderes kann der parallelen Version des Cc2s-Algorithmus passieren: Gehen wir wieder von der Situation der Abbildung 3.6a) aus. Alles läuft wie gehabt ab, bis die Targets $t_{v.1.1}$, $t_{v.1.2}$ und $t_{v.1.3}$ den Knoten $v.1.1$, $v.1.2$ und $v.1.3$ zugewiesen worden sind. Der Algorithmus versuche nun gleichzeitig $t_{v.1.1}$ und $t_{v.1.2}$ zu erfüllen. Die Arbeit, die unterhalb von $v.2$ geleistet wird, kann aber dazu führen, daß $v.1.2$ und damit auch $v.1$ den Wert $('\#' , 5)$ erhalten (Abbildung 3.6c)). Wenn unter diesen Vorbedingungen v im Verlauf weiterer Berechnungen das Target $t_v = (('\leq' , 4), 0, 2)$ bekommt, steht der Knoten $v.2$ nicht mehr als Helfer bei der Erfüllung des Targets t_v zur Verfügung.

Im Extremfall kommt der der sequentielle Cc2s-Algorithmus schnell zu einem Ergebnis, während die parallele Version erst das gesamte Spiel untersucht. Letzteres heißt für die Praxis, daß möglicherweise keine selbständige Terminierung in akzeptabler Zeit stattfindet. Es kann allerdings auch der umgekehrte Fall eintreten, daß die parallele Verarbeitung den Rechenprozeß verkürzt. In dem Spezialfall, daß bei einer sequentiellen Berechnung alle Targets erfüllt werden, untersuchen sequentieller und paralleler Cc2s-Algorithmus dieselben Knoten.

Anmerkung: Solch große Laufzeitunterschiede zwischen der sequentiellen und der parallelen Version treten bei den bisher bekannten Varianten paralleler Spielbaumsuche nicht auf. Im schlimmsten Fall allerdings durchsucht jeder bisher bekannte parallele $\alpha\beta$-Algorithmus alle Knoten bis zur vorgegebenen Tiefe, während der sequentielle $\alpha\beta$-Algorithmus lediglich den für ihn nötigen minimalen Baum (der besteht aus den Knoten, die jeder Algorithmus mindestens untersuchen muß) durchsucht. Für Anwendungen bedeutet das dann aber ebenfalls, daß eine selbständige Terminierung in vernünftiger Zeit nicht zustande kommt.

2. Verfügbarkeit von gleichzeitig bearbeitbaren Teilproblemen:
   Wenn nicht zu jedem Zeitpunkt genügend parallel verarbeitbare Teilprobleme zur Verfügung stehen, sinkt die Prozessorauslastung.

3. Kommunikationsgeschwindigkeit:
   Eine wichtige Rolle wird es spielen, wieviel Zeit eine Nachricht benötigt, um von einem zu einem anderen Prozessor zu gelangen. Die Zeit, die die Beschreibung eines Teilproblems braucht, um zu einem arbeitslosen Prozessor zu gelangen, senkt die Prozessorauslastung. Die Zeit, die eine Nachricht braucht, um einen Prozessor davon zu unterrichten, daß seine derzeitige Arbeit überflüssig ist, erhöht den Suchoverhead.

4. Zeit, die für die Verarbeitung von Nachrichten benötigt wird:
   Nachrichten, die im Prozessornetz ausgetauscht werden, müssen auch verarbeitet werden. Die Zeit, die dafür benötigt wird, steht der eigentlichen Anwendung nicht mehr für eigene Berechnungen zur Verfügung.

5. Scheduling Zeiten:
   Wir werden dafür sorgen müssen, daß jeder Prozessor mehrere Teilprobleme gleichzeitig halten und verwalten kann. Die Verwaltung der Teilprobleme kostet ebenfalls Zeit.

3.3.2 Designentscheidungen bei der Lastverteilung

Ausnutzung vs. Zerschlagung der Problemstruktur

Um die Gesamtlast im Prozessornetzwerk zu verteilen, gibt es zwei grundsätzlich gegensätzliche Ansätze. Zum einen kann man versuchen, die im jeweiligen Problem vorhandene Struktur auszunutzen. In unserem Fall hieße das, man kann versuchen, die Baumstruktur des Zweipersonen-Spiels zu nutzen und eine Einbettung des Baumes in das Prozessornetzwerk zu berechnen. Allerdings ist der entstehende Spielbaum sehr unregelmäßig und seine Form nicht vorhersehbar.

Aussichtsreicher erscheint es, die Struktur des Problems zu zerschlagen. Zu diesem Zweck läßt man eine mehr oder weniger zufällige Einbettung der Spielbaumknoten auf Prozessoren zu. Das ist zum einen durch Workstealing möglich, bei dem sich arbeitslose Prozessoren um Arbeit eines zufällig ausgewählten Arbeitgeberkandidaten bemühen. Zum anderen kann ein mit Arbeit gut ausgelasteter Prozessor Teilprobleme an zufällig ausgewählte Auftragnehmer versenden. Statt Arbeitnehmer zufällig auszuwählen, kann man die Arbeitnehmer auch anhand einer Hashfunktion bestimmen.

Der Nachteil dieser Methode ist, daß Baumkanten über mehrere Hardwarekanten gestreckt werden. Man gewinnt allerdings im allgemeinen eine für die Praxis ausreichende, gleichmäßige Auslastung der Prozessoren.

Migration vs. feste Plazierung von Knoten

Auch bei einer vom Zufall beeinflußten oder bei einer von einer Hashfunktion gesteuerten Lastverteilung muß man einige Cc2s-spezifische Besonderheiten berücksichtigen. Eine der auffallendsten und unangenehmsten Eigenschaften des Cc2s-Algorithmus ist es, daß Knoten, die erzeugt und untersucht worden sind, im Speicher des Rechners abgelegt werden müssen und nicht 'vergessen' werden dürfen. Andernfalls würde die im vorigen Kapitel vorgestellte Terminierungsargumentation nicht mehr gelten. D.h. aber auch, daß man sich beim verteilten Cc2s-Algorithmus entscheiden muß, ob Knoten, die von einem Prozessor P erzeugt wurden, auch dort bleiben sollen, oder ob man erlaubt, daß Knoten von einem Prozessor auf einen anderen verlegt werden. Man spricht dann von 'Migration'. Auf jeden Fall müssen Prozessoren, die auf einen Knoten des Prozessors P zugreifen wollen, das über P erledigen.

Möchte man Migration in ein paralleles Cc2s-Verfahren implementieren, bieten sich zwei Möglichkeiten an. Entweder migriert man ganze Teilbäume, oder man migriert einen einzelnen Knoten v und muß dann die Baumstruktur, in die v integriert war, wieder herstellen. Wir sind zu dem Schluß gekommen, daß beide Varianten der Migration zu kostspielig sind und haben uns deshalb entschlossen, auf Migration komplett zu verzichten.

Senderinitiierte Lastverteilung vs. Workstealing

Eine andere Entscheidung, die gefällt werden muß, betrifft die Frage, wie eine Arbeitgeber/Arbeitnehmer-Beziehung aufgebaut wird. Wir hatten uns zunächst dafür entschieden, einen Workstealing Algorithmus zu implementieren. Beim Workstealing müssen sich Prozessoren, die keine sinnvolle Arbeit leisten, um Arbeit bemühen. Üblicherweise senden zu dem Zweck arbeitslose Prozessoren Bitten um Arbeit in der Hoffnung ins Netz, daß ein arbeitender Prozessor die Nachricht auffängt und ein Teilproblem zurücksendet. Dieses Vorgehen bietet den enormen Vorteil, daß Prozessoren, die unter Last laufen, wenig kommunizieren müssen, und diejenigen Prozessoren, die leer laufen, viel kommunizieren. In einer frühen Implementierung des Controlled Conspiracy Number Verfahrens in einem Schachprogramm minderer Qualität erreichten wir zwar gute Speedups auf dem GCel-Rechner, mußten aber schon damals erkennen, daß es leider zu Problemen bezüglich der Verfügbarkeit verteilbarer Arbeit [LR96] kommt.

Eine andere Möglichkeit, eine Arbeitnehmer/Arbeitgeber-Beziehung aufzubauen besteht darin, verteilbare Teilaufgaben an zufällig ausgewählte Prozessoren zu senden. Man spricht hier von senderinitiierter Lastverteilung. In Zeiten, in denen wenig Last im Netz vorhanden ist, werden die zufällig verschickten Teilprobleme mit hoher Wahrscheinlichkeit zu einem arbeitslosen Prozessor gesendet. Der Vorteil dieser Methode ist, daß sich die vorhandene Gesamtlast gleichmäßig im Netzwerk verteilt und ein Prozessor P keine Probleme bei der Lastverteilung damit bekommt, daß viele Teilprobleme, die eigentlich für externe Bearbeitung geeignet wären nicht abgegeben werden können, weil die potentiell abgebbaren Knoten schon einmal auf P untersucht worden sind und bereits P fest zugeordnet sind [LR96]. Nachteilig ist die Tatsache, daß bei hoher Gesamtlast weiterhin Teilprobleme abgespalten werden (was Zeit kostet), und deren Ortswechsel das Netzwerk belasten.

3.4 Senderinitiierte parallele Cc2s-Suche

3.4.1 Grundversion

Datenstrukturen und Speicherverwaltung

Unser paralleler Cc2s-Algorithmus braucht die Möglichkeit, Teilprobleme abzugeben und Ergebnisse, die auf entfernten Prozessorknoten berechnet wurden, zu integrieren. Zu diesem Zweck wird die Rekursion, deren Verwaltung im sequentiellen Fall (Abb. 2.7, Seite 32) im wesentlichen vom Systemstack übernommen wurde, aufgehoben. Die parallele Variante wird die Verwaltung der Rekursion selber vornehmen.

Definition 3.4-12 **(Aktuelle Variante)**
Die Prozedur Cc2s(v, t_v) bearbeite einen Knoten $v \in V$ eines Spielbaums $G = ((V,E),f)$ mit Wurzel ϵ. Dann gibt es einen Pfad von der Wurzel ϵ zum Knoten v. Diese Folge von Knoten $(\epsilon = v^1, v^2, \ldots, v^d = v)$ nennen wir die *aktuelle Variante* der Prozedur Cc2s(...) zu einem Zeitpunkt τ. d nennen wir die *Ebene*, in der sich der Suchalgorithmus befindet. □

Definition 3.4-13 **(Bearbeitungszustand eines Problems)**
Sei $G = ((V, E), f)$ ein Spielbaum, und sei die Knotenfolge $(v^1, v^2, \ldots, v^d)$ eine aktuelle Variante der Prozedur Cc2s(...). Sei W die Menge der möglichen Werte und T die Menge der Targets. Dann beschreibt die Folge $((v^1, t^1, z^1), \ldots, (v^d, t^d, z^d)) \in (W \times T \times \{\text{START, REKURSION, WARTET,KEINE\_ADRESSE}\})^d$ den aktuellen Bearbeitungszustand des Problems, das gerade vom Cc2s-Algorithmus untersucht wird. Dabei sind v^1,

$\ldots, v^d$ Knoten, $t^1, \ldots, t^d$ dazugehörige Targets und $z^1, \ldots, z^d$ Zustände (im Grunde Programmzeilennummern), in denen sich der Cc2s-Algorithmus in den Ebenen 1 bis d befindet. □

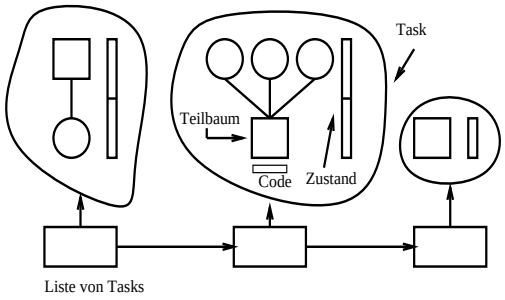


Abb. 3.7: Datenstrukturen

Jeder Prozessor ist in der Lage, viele Probleme zur gleichen Zeit zu verwalten. Um Deadlocks zu vermeiden bzw. beheben zu können, speichert ein Prozessor nicht nur die Problembeschreibungen selber ab, sondern auch deren Bearbeitungszustand.

Eine Kombination (P, B, c, S) von Problembeschreibung P, Bearbeitungszustand B, einem Erkennungscode c und einem Spielbaum G, dessen Wurzel dem Knoten der Problembeschreibung entspricht, nennen wir einen 'Task'. Mehrere Tasks werden in einer Liste verwaltet (Abbildung 3.7).

Bearbeitung eines Tasks

Es darf immer nur ein Task *aktiv* sein, d.h. gerade von einem Prozessor bearbeitet werden. Ein *Scheduler* verwaltet eine Liste von Tasks, die er zu bestimmten Zeitpunkten auswechseln darf. Er darf dies zu Beginn und zum Ende eines Cc2s-Schrittes, weil sich der Cc2s-Algorithmus des aktiven Tasks zu diesen Zeitpunkten in einem wohldefinierten Zustand befindet. Damit ergibt sich die in Abbildung 3.8 gezeigte Prozeßstruktur. Beim parallelen Cc2s-Algorithmus werden Tasks normalerweise nicht deaktiviert. Nur, wenn ein Task nicht weiterarbeiten kann, weil er nur noch auf Antworten anderer Prozessoren wartet, wird er ausgegliedert, und es wird der vorderste in der Taskliste gefundene, nicht wartende Task aktiviert.

Ein Prozessor, der ein Teilproblem zur Bearbeitung erhalten hat, ist für die korrekte Bearbeitung dieses Teilproblems verantwortlich. Er initialisiert sein Teilproblem (Abb. 3.9,

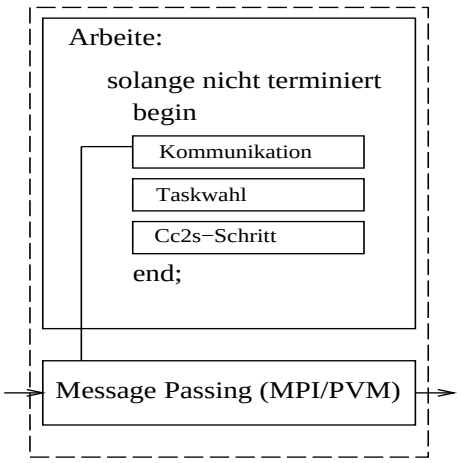


Abb. 3.8: Prozeßstruktur

Z. 1ff) und beginnt, wie im sequentiellen Fall eine Suche auf dem erhaltenen Knoten des Spielbaums.

```
bool pCc2s(node v, target t_v = ((α_v, β_v), δ_v, γ_v))
        /* Pseudorekursiver Cc2s-Algorithmus */
0       int d;
        ...
1       d := 0;
2       v^1 := v; /* Wurzel ε des Spielbaums */
3       t_v^1 := t_v;
4       z^1 := START;
5       /* Iteriere bis Wurzelstellung gelöst. */
6       while d ≥ 0 do
7           Cc2s_step();
6       return r^0;
```

Abb. 3.9: Iterationsschleife des parallelen Cc2s-Algorithmus

Die in Abb. 3.9 (Z. 6ff) dargestellte Kontrollschleife ersetzt zusammen mit den Zeilen 0a-0c,8a-8c,10b und 14a der Cc2s_step(...)-Prozedur (Abb. 3.10) die vom System verwaltete Rekursion des sequentiellen Cc2s-Algorithmus. Die fett dargestellten Zeilen sind zur sequentiellen Suchprozedur hinzugekommen.

Geschwisterknoten $v_1, \ldots, v_n$ der aktuellen Variante werden von der Prozedur Split(...) in Zeile 6 (Abb. 3.10) mit Targets versehen. Diese Knoten mitsamt den dazugehörigen

```
Cc2s_step()
        /* Es sei (v^1, ..., v^d) die aktuelle Variante, */
        /* es seien t_v^j, j ∈ {1, ..., d} die Targets in den Ebenen j und*/
        /* es seien r^j, i^j, j ∈ {0, ..., d} Hilfsvariablen, N die Anzahl der Prozessoren */
0a          if z^d = REKURSION springe zu Zeile 9.
0b          if z^d = KEINE_ADRESSE springe zu Zeile 6f.
0c          if z^d = WARTET springe zu Zeile 12b.
0d          Behandle Inkonsistenzen(v^d,t_v^d);
1           if (δ_{v^d} = 0 and γ_{v^d} ≤ 1) or |Γ(v^d)| = 0 { r^{d-1} = OK; d = d − 1; return;}
2           r^d := NOT_OK;
3           while r^d = NOT_OK do {
3b              befreie alle Nachfolger von v von ihren Aufgaben;
4               PartialExpansion(v^d, t_v^d);
5               if not OnTarget(v^d, t_v^d) { r^{d-1} = NOT_OK; d = d − 1; return;}
6               Split(v^d, t, v_1^d ... v_{|Γ'(v^d)|}); /* weist den Nachfolgern Targets zu */
6a              for i := 1 to |Γ(v^d)'| do { /* eigentlich i^d statt i*/
6b                  if (e := Hashschlüssel(v_i^d) mod N) ≠ MYID then do {
6c                      Sende PROBLEM(MYID,e,v_i^d,t_i^d,a_i^d,c);
6d                      /* Sei dabei a_i^d, falls vorhanden, die Adresse */
6e                      /* von v_i^d auf Prozessor e, sonst sei a_i^d = -1*/
6f                  }
6g              falls es ein i gibt mit a_i^d = −1 { z^d := KEINE_ADRESSE; return }
7               for i := 1 to |Γ(v^d)'| do { /* eigentlich i^d statt i*/
7b                  if Hashschlüssel(v_i^d) mod N = MYID then do {
8a                      v^{d+1} := v_i^d;t_v^{d+1} := t_i^d;
8b                      z^d := REKURSION; z^{d+1} := START;
8c                      d = d + 1;return ;
9                       w_v^d := UpdateValue(v^d);
9a                      if r^d = NOT_OK befreie alle Nachfolger von ihren Aufgaben;
10a                     if not OnTarget(v^d, t_v^d)
10b                         { r^{d-1} = NOT_OK; d = d − 1;return; }
11                      if r^d = NOT_OK break; /* springe zu Z.3 */
11a                 }
12              }
12b             if r^d := Ergebnisse_stehen_aus() { z^d := WARTET; return ;}
13          } /* while ... */;
14a         r^{d-1} = OK; d = d − 1; return;
```

Abb. 3.10: Pseudo-Rekursion des parallelen Cc2s-Algorithmus

Targets beschreiben Teilprobleme, die gleichzeitig bearbeitet werden können. Bearbeite nun ein Prozessor P einen Task T. In den Zeilen 6a bis 6f der Prozedur Cc2s_step(...) (Abb. 3.10) werden Teilprobleme als Arbeitspakete verschickt, wobei der jeweilige Zielprozessor gemäß einer Hashfunktion bestimmt wird. Dabei liefert der Hashschlüssel eines Knotens v_i ($1 \leq i \leq n$) modulo der Prozessoranzahl den Zielprozessor. Falls der Zielprozessor eines Arbeitspaketes mit P identisch ist, wird dieses Paket nicht verschickt, sondern in den Zeilen 7-12 der Cc2s_step-Prozedur auf Prozessor P weiterverarbeitet.
In Zeile 12b (Abb. 3.10) wartet der Algorithmus auf ausstehende Ergebnisse. Die Prozedur Ergebnisse_stehen_aus() liefert *OK* zurück, wenn alle bis zu dem Zeitpunkt des Aufrufs eingegangenen Teilergebnisse *OK* waren, andernfalls gibt sie *NOT_OK* zurück.
Zu bemerken ist noch, daß Knoten, die verschickt werden, im Gesamtsystem doppelt vorliegen, einmal auf der Empfängerseite als vollständiger Knoten mitsamt Nachfolgerlisten etc. und einmal als 'kleiner' Knoten auch auf der Senderseite. Dort ist zum einen vermerkt, wo sich der entfernte Knoten befindet, zum anderen ist dort aber auch sein Wert ge-cached. Dadurch wird Kommunikation eingespart.

Beginn von Auftraggeber/Auftragnehmer-Beziehungen

Im initialen Zustand sind alle Prozessoren arbeitslos. Ein ausgezeichneter Prozessor P_0 bekommt die Wurzelstellung. Er führt die Prozedur DetermineMove(...) aus, wobei er die Teilprobleme, die er an Nachfolgern der Wurzel generiert, nicht selber bearbeitet, sondern sie an andere Prozessoren verschickt. Die Zielprozessoren werden mit Hilfe einer Hashfunktion ausgewählt.
Erhält ein Prozessor e eine Teilproblem-Nachricht PROBLEM=(s,e,v,t_v,a,c), führt er die in Abbildung 3.9 (Zeilen 1-4) beschriebene Initialisierung durch und erzeugt einen neuen Task. Dabei ist s der Sender des Problems und e der Empfänger. Der Knoten v ist die Wurzel des zu durchsuchenden Spielbaums, t_v das dazugehörige Target. Falls bereits festgelegt, ist a die Adresse von v beim Empfänger e. Der Code c ermöglicht eine eindeutige Identifizierung des Problems und eine eindeutige Identifizierung des Tasks auf Prozessor s, aus dem das Teilproblem (v, t_v) entstanden ist.
Falls v zum ersten Mal von e bearbeitet wird, schickt e eine Nachricht ADRESSE=(e,s,-c,a) an s zurück, damit sich s bei späteren Bearbeitungen des Knotens v auf die Adresse a beziehen kann. Im Gegensatz zum Code c bleibt a immer konstant. Es ist eine neue Auftraggeber/Auftragnehmer-Beziehung entstanden.

Da Teilprobleme des Gesamtproblems senderiniitiert verschickt werden (Abb. 2.7, Z.6a-6f), kann es passieren, daß Prozessoren mehrere Teilprobleme gleichzeitig zugesendet bekommen. Es kann dabei auf einem beliebigen Prozessor P zu keinen Deadlocks kommen,

weil Spielbäume keine Kreise haben und Tasks, bei denen sich der Cc2s-Algorithmus in der letzten Ebene im Zustand WARTET befindet, der Prozessor entzogen wird. Zu welchem Zeitpunkt ein Teilproblem bearbeitet wird, hängt allerdings von den übrigen sich auf P befindlichen Tasks ab.

```
Kommunikation
1     /* Empfange Botschaften*/
2     /* p sei die eigene Prozessornummer */
3     while Botschaft mit Identifizierungscode c empfangen {
4         if PROBLEM(s,p,v,t,a,c) empfangen {
5             erzeuge neuen Task;
6             initialisiere Problem gemäß Abb. 3.9, Z. 1-4;
7             merke s,t,c;
8             if a = −1 sende ADRESSE(p,s,c,a'); /* a' Adresse von v*/
9         }
10        if ADRESSE(e,p,c,a) empfangen {
11            finde Task T und Knoten v, die zu c gehören;
12            Ordne die Adresse a von v auf e dem Knoten v auf Prozessor p zu;
13        }
14        if ANTWORT=(s,e,x,ok,c) empfangen {
15            finde Task T, Knoten v und Ebene d, die zu c gehören;
16            setze den Wert von v zu x; UpdateValue(v);
17            if ok=true
18                markiere v in Task T, Ebene d mit OK;
19            else
20                markiere v in Task T, Ebene d mit NOT_OK;
21                von Ebene d bis zur Ebene d', in der sich T befindet,
22                befreie alle Nachfolger von ihren Aufgaben und setze d' := d.
23        }
24        if STOP=(s,e,c) empfangen {
25            finde Task T, der zu c gehört;
26            für alle Knoten der aktuellen Variante des Tasks T
27                befreie alle Nachfolger von ihren Aufgaben;
28            beende Task T;
29        }
30    }
```

Abb. 3.11: Kommunikation

Ende einer Auftraggeber/Auftragnehmer-Beziehung

Es gibt zwei grundsätzlich verschiedene Arten, auf die eine Auftraggeber/Auftragnehmer-Beziehung beendet werden kann.

Zum einen kann sie vom Auftragnehmer beendet werden. In diesem Fall hat der Auftragnehmer s seine Arbeit an einem Task T beendet und sendet eine ANTWORT=(s,e,x,ok,c) an seinen Arbeitgeber e. Er teilt ihm mit, ob das Target, das er zugewiesen bekommen hat, erfüllt wurde ($ok = true$), den möglicherweise veränderten Wert x des Startknotens und den Code c, mit dessen Hilfe der Arbeitgeber die Antwort einordnen kann. Der Empfänger der Antwort (also der Arbeitgeber) muß zum einen den Ort wiederfinden, an dem die Antwort eingefügt werden muß (d.h. die Knotenadresse im Speicher), und er muß mit Hilfe des Wiedererkennungscodes c herausfinden, zu welchem Task die Antwort von e gehört. Schlimmstenfalls gibt es den zur Antwort gehörenden Task gar nicht mehr. Die empfangene Antwort wird dann ignoriert.

Auch ein Arbeitgeber kann eine Auftraggeber/Auftragnehmer-Beziehung beenden. Wenn in Zeile 8 der Cc2s_step-Prozedur (die einen Task T bearbeite) ein Nachfolger eines Knotens v mit NOT_OK antwortet, werden die aktiven Teilprobleme unterhalb von v abgebrochen (entweder in Zeile 3 oder in Zeile 11). Entweder beendet Cc2s_step(...) dann die Arbeit an v (zumindest vorläufig), oder t_v wird neu gesplittet. In beiden Fällen wird für jeden Nachfolger von v je eine STOP=(s,e,c) Nachricht an die Prozessoren versendet, die die Teilprobleme bearbeiten, deren Wurzeln die Nachfolger von v sind. Für den Fall, daß Cc2s_step(...) den Knoten v nicht verläßt, werden möglicherweise einige gerade deaktivierte Teilprobleme sofort wieder neu aufgesetzt. Aufgrund von Nachrichtenverzögerungen kann es dabei vorkommen, daß eine der Adressen der Nachfolger von v noch nicht beim Arbeitgeber angekommen ist. Die Weiterbearbeitung des Tasks T muß dann warten, bis die ADRESSE-Nachricht angekommen ist (Zeile 6g, Abb. 3.10).

Auftretende Inkonsistenzen von Werten

In Abschnitt 1.8 auf S.22ff wurden die Begriffe 'ungeklärt', 'widersprüchlich' und 'unterstützend' für zwei Werte eingeführt. Sei ein Knoten v gegeben. Seien w_v dessen Wert und w_t die Wertkomponente eines Targets t_v, das für v gebildet wurde. Beim sequentiellen Cc2s-Algorithmus ist nach einer Splitoperation an einem Knoten garantiert, daß der Wert dieses Knotens den Werten der Nachfolger des Knotens nicht widerspricht. Wenn der Cc2s-Algorithmus einen Knoten v mit Target t_v besucht, ist außerdem garantiert, daß der Wert w_v von v die Wertkomponente w_t des Targets t_v unterstützt. Diese Eigenschaften machten im vorigen Kapitel dieser Arbeit unsere Korrektheits- und Terminierungsbewei-

se möglich.

Sei P_i nun der Prozessor, der für die Bearbeitung eines Knotens v zuständig ist. Sei $w_v = (a_v, z_v)$ der Wert von v auf P_i. Ferner sei $t_v = ((\alpha_v, \beta_v), \gamma_v, \delta_v)$ das Target, das ein Prozessor P_j an P_i für den Knoten v gesendet hat. Wir gehen, wie weiter vorne beschrieben, davon aus, daß auf P_j eine (kleine) Kopie des Knotens v vorliegt. In dieser Kopie stehen lediglich ein Verweis auf v auf P_i und ein Wert. Der Wert wird in erster Linie benötigt, um am Vorgänger von v die Splitprozedur ausführen zu können. Müßte man für jeden Aufruf der Splitprozedur erst die Werte aller Nachfolger einsammeln, würde das zu unnötiger Belastung des Netzwerkes führen. Im Prinzip handelt es sich bei dem Speicherplatz, der für die Kopie von v auf P_j benötigt wird um einen kleinen Cache. Im parallelen Fall kann es nun passieren, daß bei P_i ein Arbeitsauftrag ankommt, bei dem das Target des zugesendeten Teilproblems nicht von dem Wert von v unterstützt wird. Der Grund ist dann natürlich, daß P_i und P_j unterschiedliche Bewertungen von v zugrunde gelegt haben. Wir sprechen in diesem Fall von einer aufgetretenen 'Inkonsistenz'. Im folgenden untersuchen wir alle prinzipiell möglichen Inkonsistenzen zwischen der Wertkomponente $w_t := (\alpha_v, \beta_v)$ des Targets t_v und dem Wert w_v. Glücklicherweise kann man, wenn man einen Knoten v besucht, alle Inkonsistenzen zwischen v und seinen Nachfolgern lokal beseitigen.

Folgende Fälle, die wir oBdA. wieder nur für MAX-Knoten betrachten, können beim Besuch eines Knotens v auftreten:

1. Das Target t_v fordert, ausgedrückt durch seine Wertkomponente w_t, eine schärfere Schranke, als w_v zum Ausdruck bringt.

   (a) Es sei v ein MAX-Knoten mit dem Wert $w_v = (\text{'}\geq\text{'}, z_v)$ auf Prozessor P_i. Dann ist $w_t = (\text{'}\geq\text{'}, \beta_v)$ mit $\beta_v > z_v$. Wir wissen nun, daß das Attribut a_v von v immer schon '$\geq$' gewesen ist, und daß Nachfolger von v den Wert (genauer den Zahlwert z_v von v) von v allenfalls größer gemacht haben können. Also wurde v auf P_j direkt bewertet, und P_j ist lokal zu dem Schluß gekommen, daß der Wert von v $(\text{'}\geq\text{'}, \beta_v)$ mit $\beta_v > z_v$ sein muß. Deshalb hat P_i für v ein Target $t_v = ((\text{'}\geq\text{'}, \beta_v), \ldots, \ldots)$ bekommen. Wenn v nun auf P_i bewertet würde, bekäme v natürlich dort dieselbe direkte Bewertung, die dann noch den Werten der Nachfolger von v angepaßt werden müßte. Es genügt deshalb, den Zahlwert von v auf β_v hochzusetzen (d.h., v bekommt auch auf P_i den Wert $(\text{'}\geq\text{'}, \beta_v)$) und auf P_i UpdateValue(v) aufzuführen.

   (b) Auf ähnliche Weise kann es passieren, daß ein MAX-Knoten v auf einem Prozessor P_i einen Wert $(\text{'}\leq\text{'}, z_v)$ besitzt und das Target $t_v = ((\text{'}\leq\text{'}, \beta_v)$,-

$\ldots,\ldots)$ mit $z_v > \beta_v$ zugesendet bekommt. Dann wurde vorher die kleine Kopie von v auf dem Prozessor P_j, von dem aus (v, t_v) gesendet wurde, mit einem Wert der Form $(\text{'\#'}, \beta_v)$ versehen, und es reicht nun aus, entweder auf dem Prozessor P_i auch eine direkte Bewertung von v durchzuführen, so daß v danach einen Wert der Form $(\text{'\#'}, \beta_v)$ hat, oder auf P_i den Wert von v von P_j zu übernehmen. Natürlich muß auch hier wieder UpdateValue(v) aufgerufen werden.

In beiden Fällen kann der Cc2s-Algorithmus weiterarbeiten, als hätte es keine Inkonsistenzen gegeben.

2. Die Werte w_t und w_v sind widersprüchlich:

   (a) Sei $w_v = (\text{'\#'}, z_v)$. Dann senden wir eine Antwort mit dem Wert w_v und NOT_OK an P_j. P_j wird sich w_v merken. Der Fall, daß P_j t_v verschickt hatte, weil P_j v direkt bewertet hatte, kann nun nicht mehr vorkommen. Falls P_j t_v verschickt hatte, weil sich w_v auf P_i geändert hatte, aber P_j dies nicht schnell genug mitbekommen hatte, wird P_j nach endlicher Zeit den richtigen Wert von v erfahren und danach nicht mehr das unpassende Target t_v versenden.

   (b) Seien $w_v = (\text{'}\geq\text{'}, z_v)$ und $w_t = (\text{'}\leq\text{'}, \beta_v)$ mit $\beta_v < z_v$.
   Da w_v das Attribut '$\geq$' besitzt, muß sein erster Wert $(\text{'}\geq\text{'}, z)$ mit $z \leq z_v$ gewesen sein. Der Wert w_v (genauer sein Zahlwert z_v) kann durch die Nachfolger von v nämlich allenfalls größer gemacht worden sein. Wenn P_j also ein Target mit Wertkomponente w_t verschickt hat, muß v auf P_j bereits einen Wert $(\text{'\#'}, z'_v)$ mit $z'_v < \beta_v < z_v$ besitzen. Man kann also schließen, daß eine direkte Bewertung von v den Wert $(\text{'\#'}, z'_v)$ liefert und daß mindestens 1 Nachfolger von v zeigt, daß es eine Hülle unter v gibt, deren Minimaxwert $\geq z_v$ ist. Wir setzen deshalb w_v auf P_i zu $(\text{'\#'}, z_v)$ (das ist das, was auch eine direkte Bewertung plus UpdateValue(v) liefern würde) und senden an P_j eine Antwort mit dem Wert w_v für v und NOT_OK. Sobald die Antwort P_j erreicht, ist die Widersprüchlichkeit der Werte von v auf P_i und P_j beseitigt.

   (c) Seien $w_v = (\text{'}\leq\text{'}, z_v)$ und $w_t = (\text{'}\geq\text{'}, \beta_v)$ mit $\beta_v > z_v$.
   Da w_t nur aus älteren Werten von v (auf P_i) und einer zusätzlichen direkten Bewertung von v auf P_j zustande gekommen sein kann und w_v durch die Nachfolger von v (auf P_i) nur größer geworden sein kann, kann dieser Fall nicht eintreten.

3. w_t ist ungeklärt zu w_v, und w_v ist ungeklärt zu w_t:

(a) Sei w_v von der Form $(\text{'\#'}\ , z_v)$. Da ein '#' -Wert nicht ungeklärt sein kann zu einem anderen Wert, kann dieser Fall nicht eintreten.

(b) Seien $w_v = (\text{'}\leq\text{'}\ , z_v)$ und $w_t = (\text{'}\geq\text{'}\ , \beta_v)$ mit $\beta_v < z_v$. In diesem Fall kann nur eine direkte Bewertung von v auf P_j auf P_j zu einem Wert $(\text{'\#'}\ , z)$ mit $z \geq \beta_v$ geführt haben. Wie groß der Wert von v auf P_j ist, kann man aus dem Target nicht erkennen. Die Inkonsistenz wird beseitigt, indem man entweder v auf P_i ebenfalls direkt bewertet, oder indem man den Wert von v, der auf P_j ermittelt wurde, in w_v auf P_i einsetzt und danach UpdateValue(v) aufruft.

(c) Sei $w_v = (\text{'}\geq\text{'}\ , z_v)$ und $w_t = (\text{'}\leq\text{'}\ , \beta_v)$ mit $\beta_v > z_v$. Auch in diesem Fall kann nur eine direkte Bewertung von v auf P_j zu einem Wert $(\text{'\#'}\ , z)$ mit $z \leq \beta_v$ geführt haben. Wie groß der Wert von v auf P_j ist, kann man aus dem Target nicht erkennen. Die Inkonsistenz wird beseitigt, indem man entweder v auf P_i ebenfalls direkt bewertet, oder indem man den Wert von v, der auf P_j ermittelt wurde, in w_v auf P_i einsetzt und danach UpdateValue(v) aufruft.

```
Behandle Inkonsistenzen
      /* Sei v im folgenden ein MAX-Knoten. Sei w_v = a_v, z_v der Wert von v */
      /* und w_t die Wertkomponente des Targets t_v = ((α_v, β_v)γ_v, δ_v).*/
1   if (w_v = ('≥' , z_v) and  t_v = (('≥' , β_v), ... , ... ) mit β_v > z_v) z_v := β_v;
2   if (v ist ein MAX-Knoten and  w_v = ('≤' , z_v) and
3         t_v = (('≤' , β_v), ... , ... ) mit β_v < z_v)
4      { w_v := Evaluate(v, −∞, β_v);a_v := '#' ; }
5   if die Wertkomponente w_t von t_v widerspricht w_v {
6      if w_v = ('≥' , z_v) und w_t = ('≤' , β_v) mit β_v < z_v
7         w_v := ('#' , z_v);
8   }
9   if w_t ist ungeklärt zu w_t und w_t ist ungeklärt zu w_v {
10        { w_v := Evaluate(v, z_v , ∞);a_v :='#' ;}
11  UpdateValue(v);
12  if (Splitoperation nicht möglich)
13     verlasse Cc2s_step(... );
14     Sende Antwort mit NOT_OK und w_v;
    /* analog für MIN-Knoten ... */
```

Abb. 3.12: Behandlung von Inkonsistenzen

Außerdem gibt es rein temporäre Inkonsistenzen, die dadurch auftreten, daß Nachrichten ihr Ziel nicht zeitgerecht erreichen. Diese braucht man aber nicht weiter zu betrachten, da sie, sofern sie sich nicht sowieso als irrelevant erweisen, nach endlicher Zeit aufgelöst werden.

Beispiele für Kommunikation

Abbildung 3.13 zeigt die parallele Bearbeitung zweier Knoten $v.2$ und $v.3$. Der Wert von v ist $('\leq' , 5)$, und es soll gezeigt werden, daß dies auch mit der Sicherheit einer Resttiefe vier und einer Conspiracy Number zwei so ist. Alle Nachfolger von v bekommen deshalb das Target $(('\leq' , 5), 3, 2)$. Da für $v.1$ schon feststeht, daß er das Target erfüllt, sind noch die Berechnungen für $v.2$ und $v.3$ zu erledigen. Der Hashschlüssel von $v.2$ zeige, daß $v.2$ auf den Prozessor $P2$ gehört. $P1$ sendet deshalb das Teilproblem $(P1, P2, (('\leq' , 5), 3, 2), -1, c)$ an $P2$. Da auf $P2$ der Knoten $v.2$ zuvor noch nicht untersucht worden ist, wird $v.2$ angelegt und bekommt eine Adresse a. Diese Adresse wird sofort an $P1$ zurückgeschickt. Dadurch kann der Knoten $v.2$ bei einer weiteren, späteren Bearbeitung wiedergefunden werden. Nach Abarbeitung des Teilbaums mit Wurzel $v.2$ sendet $P2$ den ermittelten Wert x und OK an $P1$ zurück. Das Target $t_{v.2}$ wurde also erfüllt.

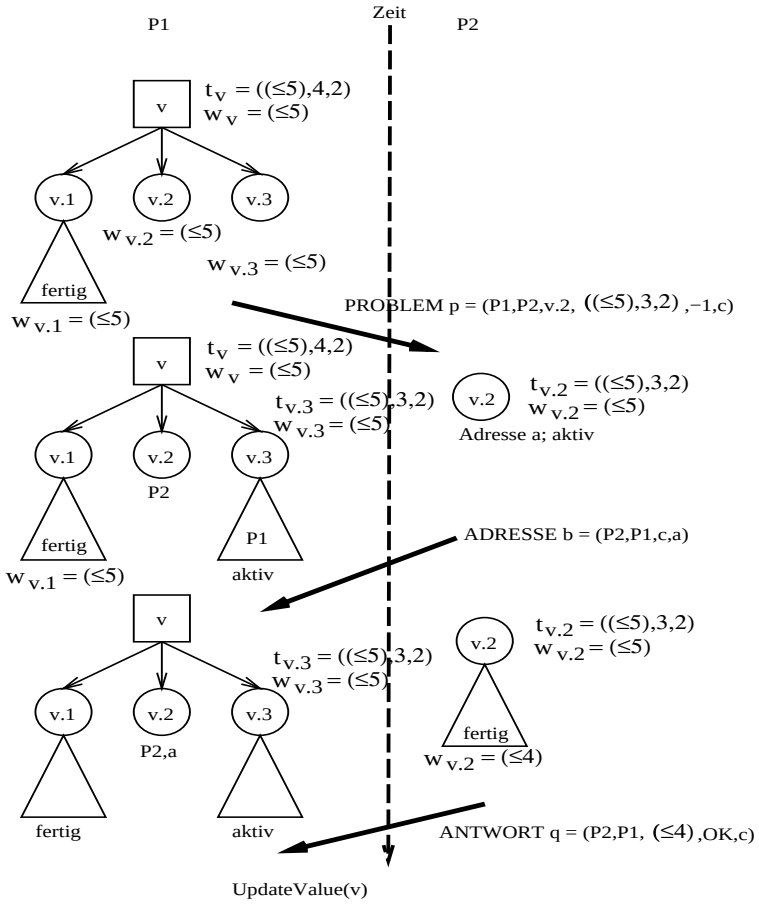


Abb. 3.13: Parallelität

In Abbildung 3.14 haben die Knoten $v.1$ und $v.2$ ein Target $((\text{'}\geq\text{'}, 4), 5, 2)$ erhalten. $v.2$ muß von Prozessor $P2$ bearbeitet werden und wurde dort auch schon früher einmal untersucht. Deshalb wird das Problem $(P1, P2, ((\text{'}\geq\text{'}, 4), 5, 2), a, c)$ an $P2$ gesendet. Dabei ist a die Adresse, wo sich bei $P2$ der Knoten $v.2$ befindet. $P2$ fängt an, $v.2$ zu bearbeiten. Gleichzeitig arbeitet $P1$ unterhalb von $v.1$. $P1$ kommt recht schnell zu dem Schluß, daß $t_{v.1}$ nicht erfüllt werden kann und daß der Wert von $v.1$ $(\text{'\#'}, 3)$ ist. Deshalb sendet $P1$ eine Stopnachricht an $P2$, um $P2$ mitzuteilen, daß er ein möglicherweise nicht mehr wichtiges Teilproblem bearbeitet. Nachdem $P2$ seine Arbeit an $v.2$ abgebrochen hat und $P1$ ein neues Target für $v.2$ erzeugt hat, sendet $P1$ eine neue Arbeitsaufforderung an $P2$.

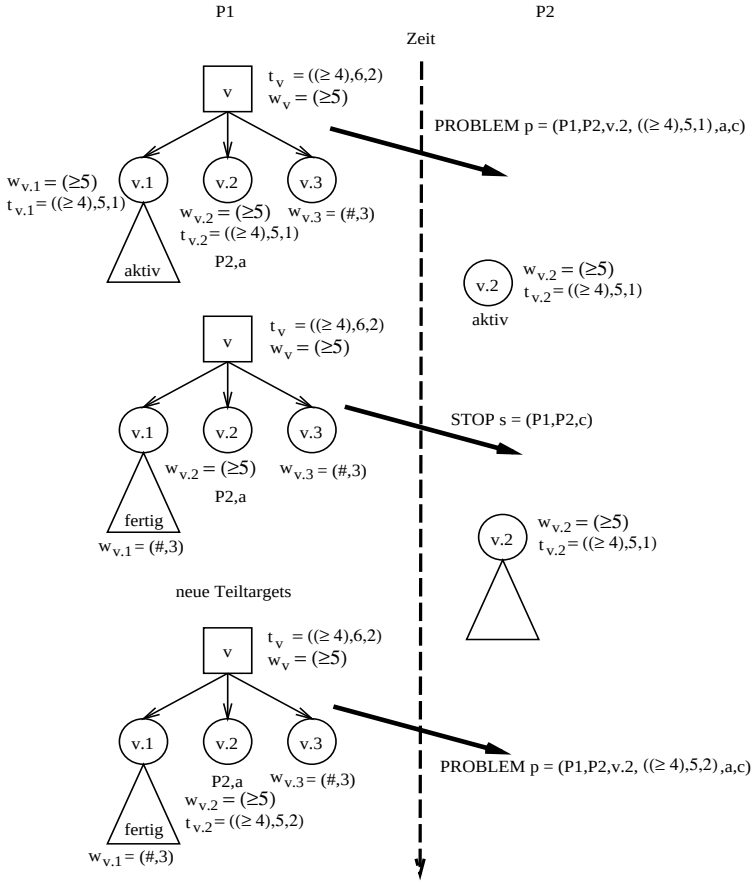


Abb. 3.14: Parallelität

3.4.2 Abgaben von Bewertungen

Bisher haben wir nur solche Knoten extern bearbeiten lassen, die von der Prozedur Cc2s_step(...) besucht werden. In der Prozedur PartialExpansion(...) ist noch Potential zur

Parallelisierung vorhanden, das wir ebenfalls nutzen.

Wenn die Prozedur PartialExpansion(...) einen Knoten v expandiert und wenn v ein ALL-Knoten ist, müssen alle Nachfolger bewertet werden. Auch diese Knoten kann man gleichzeitig bewerten lassen, indem man sie entsprechend ihrer Hashwerte ins Netzwerk verschickt. Ebenso kann man an CUT-Knoten alle Nachfolger gleichzeitig bewerten, wenn man schon weiß, daß der Knoten v sein Target t_v nicht erfüllt. Sei i die Anzahl der Nachfolger von v, die schon bewertet worden sind (vgl. Abb. 2.14 Zeile 4, bzw. Zeile 18 auf S. 40).

Unter folgenden Bedingungen verschickt der parallele Cc2s-Algorithmus einen Knoten v mit Target $t_v = ((a_v, z_v), \delta_v, \gamma_v)$:

- falls v ein ALL-Knoten ist und $i > 3$ ist (d.h. die ersten drei Nachfolger von v schon bewertet wurden),

  oder

  falls v ein CUT-Knoten ist und der Algorithmus sich in den Zeilen 27-31 der Prozedur PartialExpansion(...) befindet

  oder

  falls v ein CUT-Knoten ist und $i > \gamma + 3$ ist (d.h., es wurden schon 3 Nachfolger mehr bewertet, als mindestens zur Erfüllung von t nötig waren)

- und falls die Anzahl der Tasks < 200 ist

- und falls v bewertet werden muß (vgl. Abb. 2.15, S.41).

Da die Teilbäume unter den Nachfolgern von v jeweils nur aus einem Knoten bestehen, gibt es keinen Grund, sie schon auf den entfernten Prozessoren zu fixieren. Damit erhalten wir einen nützlichen Freiheitsgrad: Wenn die Gesamtlast im Netzwerk groß ist, kann ein Nachfolger des Knotens v von dem Prozessor bearbeitet werden, bei dem v liegt, sonst kann der Knoten auch verschickt werden. Um lokal auf einem Prozessor P die Last des Gesamtnetzes abzuschätzen, benutzen wir eine einfache Heuristik: Wir gehen bei einem Prozessor P davon aus, daß die Gesamtlast im Netz groß ist, wenn die Anzahl der Tasks lokal bei P größer oder gleich 200 ist.

3.4.3 Verteilte Transpositionstabelle

Der sequentielle Algorithmus bedient sich einer Transpositionstabelle, also einer Hashtabelle für Knoten, in der zu bereits besuchten Stellungen des Spielbaums die dazugehörigen Ergebnisse eingetragen werden. Auf diese Weise verhindert man in vielen Fällen, daß

Stellungen, die durch Zugumstellungen im Suchbaum entstehen, doppelt untersucht werden.

Jeder Prozessor im verteilten System von N Prozessoren hält eine Transpositionstabelle der gleichen Größe M. Diese lokalen Teiltabellen werden logisch als eine große Tabelle mit $M \cdot N$ Einträgen betrachtet. Die Hashfunktion liefert also einen Index $i(x) \in \{0 \ldots N \cdot M - 1\}$. Hat eine Stellung v im Spielbaum den Hashwert

$$h(v) = (i(v), l(v)) \in \{0 \ldots N \cdot M - 1\} \times \{0,1\}^{32},$$

so wird der Index i(v) aufgespalten in eine Prozessornummer

$$p(v) := i(v) \bmod N$$

und einen Index in die lokale Hashtabelle von $p(v)$

$$i_{p(v)} := \lfloor \frac{i(v)}{N} \rfloor \bmod M.$$

Da jeder Prozessor gemäß seines Hashwertes in das Prozessornetzwerk eingebettet wird, ist es leicht, globale Zugumstellungen zur Stellung v mit Hilfe der globalen Transpositionstabelle zu finden. Man benötigt lediglich Zugriffe auf die lokale Hashtabelle desjenigen Prozessors, der v bearbeitet.

3.5 Experimentelle Leistungsbewertung

Dieses Kapitel besteht aus vier Teilen. Zunächst versuchen wir die Leistung des Gesamtsystems einzuordnen, indem wir Ergebnisse des parallelen Algorithmus in der Praxis, d.h. auf Schachturnieren, und sein Verhalten auf dem BT2630 Test präsentieren.

Danach stellen wir die nötigen Begriffe vor, die wir brauchen, um die Effizienz unseres verteilten Algorithmus zu beurteilen. Den vierten Abschnitt bilden die Effizienzmessungen selber.

3.5.1 Turniere

Im Juli 2000 gewann P.ConNerS (die parallele Version von ConNerS) das 10. Lippstädter Großmeisterturnier, das im Jahr 2000 drittstärkste Turnier Deutschlands. Zum allerersten Mal weltweit konnte damit ein Computerprogramm ein internationales FIDE Großmeisterturnier (Kategorie 11) gewinnen. In einem Teilnehmerfeld von 11 menschlichen Schachmeistern mußte sich P.ConNerS nur dem ehemaligen Jugendweltmeister Slobodjan und dem erfahrenen englischen Großmeister Speelman geschlagen geben. P.ConNerS gewann das Turnier mit 6 Siegen, 3 Unentschieden und 2 Niederlagen.

Die Gegner des Programms hatten eine durchschnittliche ELO-Zahl von 2522, und P.ConNerS erspielte sich eine Turniererfolgszahl von 2660 ELO-Punkten. Das Programm hat sich damit den Ruf eines Weltklasseprogramms erworben.

Zwar hatten auch in früheren Jahren immer mal wieder Programme menschliche Schachgroßmeister geschlagen, es handelte sich aber immer entweder um einzelne Partien oder Schnell- oder Blitzschachturniere. In Lippstadt spielten die menschlichen Gegner von P.ConNerS unter für sie optimalen Bedingungen.

Eine ganze Reihe von Zeitungs- und Fachzeitschriftenartikeln belegen, daß es sich bei diesem Erfolg für die Schachspieler um eine zu diesem frühen Zeitpunkt unerwartete Sensation handelte [BILD00] [CSS00] [HS00] [ICC00] [IX00] [LS00] [NW00] [PAT00] [S00] [SCHM00] [CB00] [WT00].

Davor war P.ConNerS erst viermal auf offiziellen Turnieren aufgetreten. Bei seinem ersten Auftritt 1998 wurde das Programm vierter auf dem 7. Internationalen Paderborner Computerschachturnier, 1998 konnte P.ConNerS das 8. Internationale Paderborner Computerschachturnier sogar gewinnen, wobei es sich dabei gegen Weltklasseprogramme wie Nimzo und Shredder (Weltmeister 1999, PC-Weltmeister 1998+2000) durchsetzen konnte. Enttäuschend war lediglich der Auftritt bei dem Computerschach-Weltmeisterschaft 1999. Programmfehler bei der Portierung auf einen neuen Rechner und Fehler in der Hardware machten zu diesem Zeitpunkt ein gutes Abschneiden unmöglich. Der vierte Platz beim 9. Int. Paderborner Computerschachturnier ist ähnlich hoch zu bewerten, wie der 1. Platz 1999.

Jahr	Turnier	Rang
2000	10. Lippstädter Großmeisterturnier Kat.11/12	1
2000	9. Int. Paderborner Computerschachturnier	4
1999	9. Computerschach-Weltmeisterschaft	18
1999	8. Int. Paderborner Computerschachturnier	1
1998	7. Int. Paderborner Computerschachturnier	4

Tab. 3.1: Alle Turnierteilnahmen von P.ConNerS

3.5.2 BT2630

Eine weitere Möglichkeit, einen Eindruck von der Spielstärke eines Programms zu bekommen, sind Testreihen mit ausgewählten Teststellungen. Auf dem von uns bevorzugten BT2630 Test erreicht P.ConNerS mit 2589 (Pseudo)-ELO-Punkten ebenfalls ein von anderen Programmen unerreichtes Ergebnis. Erläuterungen zum BT2630-Test siehe Kap.2.4

#Prozessoren	BT2630-Elo
159	2586
79	2589
39	2565
19	2541
9	2470
3	2427
2	2400
1	2402

Tab. 3.2: Ergebnisse auf dem BT2630-Test

3.5.3 Definitionen

Um die Güte eines parallelen Algorithmus zu bewerten, sollte er mit der bestmöglichen sequentiellen Methode verglichen werden. Man betrachtet den *Speedup* (die Beschleunigung), die das parallele Verfahren gegenüber dem sequentiellen Verfahren erreicht. Weil der beste Spielbaumsuchalgorithmus nicht bekannt ist, beschränken wir uns darauf, den parallelen Cc2s-Algorithmus mit dem sequentiellen, im vorigen Kapitel vorgestellten Cc2s-Algorithmus zu vergleichen. Immerhin handelt es sich dabei sicherlich um den bestbekannten Conspiracy Number Suchalgorithmus.

Der Speedup eines Problems P ergibt sich aus der Bearbeitungszeit $t_1(P)$ des sequentiellen Algorithmus geteilt durch die Bearbeitungszeit $t_n(P)$ des parallelen Algorithmus mit n Prozessoren.

Für eine Menge $\mathcal{P}$ von Problemen definieren wir den Speedup folgendermaßen:

$$SPE(n) = \frac{\sum_{p \in \mathcal{P}} t_1(P)}{\sum_{p \in \mathcal{P}} t_n(P)}.$$

Mit Hilfe des Speedups ist auch die Effizienz EFF(n) definiert:

$$EFF(n) = SPE(n)/n.$$

Sei $w_n(P)$ die Summe der Zeiten, die n Prozessoren benötigen, um Tasks auszuwählen und um alle Aufrufe ihrer Baumsuchprozesse 'pCc2s' durchzuführen. Die Zeiten, in denen Prozessoren nichts anderes tun, als auf Antworten zu warten, seien in $w_n(P)$ nicht enthalten. Dann ist die durchschnittliche Auslastung definiert als

$$LOAD(n) = 100 \cdot \frac{\frac{1}{n} \cdot \sum_{p \in \mathcal{P}} w_n(P)}{\sum_{p \in \mathcal{P}} t_n(P)}.$$

Die Prozessorauslastung gibt also das Verhältnis von reiner Arbeitszeit zur Gesamtbearbeitungszeit an. Allerdings kann sich darin auch vom Betriebssystem verbrauchte Zeit, sowie Nachrichtenverarbeitungszeit und Task-Schedulingzeit verstecken. Eine gute durchschnittliche Prozessorauslastung garantiert nur, daß die Prozessoren selten arbeitslos sind. Sie gibt auch nicht an, wie sinnvoll die durchgeführten Arbeiten waren.

Um letzteres abschätzen zu können, definieren wir den Suchoverhead.

$$SOVD(n) = 100 \cdot (\frac{\sum_{p \in \mathcal{P}} k_n(P)}{\sum_{p \in \mathcal{P}} k_1(P)} - 1)$$

Der Suchoverhead gibt an, wieviele Knoten der parallele Algorithmus im Verhältnis zur sequentiellen Version untersucht hat. Hierbei zählen wir die Gesamtknotenzahlen, also auch diejenigen Knoten, die von den Tiefe-2 $\alpha\beta$-Suchen erzeugt werden, die in der Cc2s-Routine als Bewertung eingesetzt werden.

Zusätzlich definieren wir die Leistung wie folgt;

$$LSTG(n) = \frac{\sum_{p \in \mathcal{P}} k_n(P)}{\sum_{p \in \mathcal{P}} t_n(P)}$$

Die Leistung gibt an, wieviele Knoten der parallele Algorithmus im Durchschnitt über alle Probleme pro Sekunde erzeugt. Wir haben uns für diese Definition entschieden, weil dadurch der Speedup in zwei Komponenten aufgeteilt wird.

Der Speedup SPE(n) ist gleich

$$100 \cdot (\frac{LSTG(n)}{LSTG(1)} / (SOVD(n) + 100)).$$

Wenn der Suchoverhead niedrig ist und die Leistung von n Prozessoren ungefähr der n-fachen Leistung eines Prozessors entspricht, ist auch der Speedup hoch. Wenn der Speedup niedrig ist, kann es dafür im folgenden zwei Gründe geben. Zum einen kann es sein, daß die Auslastung der Prozessoren niedrig ist, oder der parallele und der sequentielle Algorithmus generieren unterschiedlich viele Knoten. Wir werden beide Probleme eingehender betrachten.

Erzeugung von Testinstanzen

Um eine Geschwindigkeitssteigerung von n Prozessoren gegenüber einem einzelnen zu messen, ist es erforderlich, die Zeiten zu messen, die die Algorithmen brauchen, um ein fest vorgegebenes Arbeitspaket abzuarbeiten. Die Arbeit, die benötigt wird, um auf einer vorgegebenen Schachstellung eine fest vorgegebene Sicherheit zu garantieren, bietet sich als ein Arbeitspaket an. Nun ist die Bearbeitungszeit, die der Cc2s-Algorithmus benötigt,

um eine fest vorgegebene Sicherheit zu erreichen von Stellung zu Stellung sehr unterschiedlich. Deshalb geben wir für jede einzelne Stellung eine zu erreichende Sicherheit vor. Dadurch besteht eine Menge $\mathcal{P}$ von Teststellungen aus dem Produkt $S \times T$, einer Menge von Stellungen S und einer Menge von Sicherheiten T. Die Sicherheiten werden so gewählt, daß das parallele Programm eine Sicherheit in einem vorgegebenem Zeitrahmen erbringt und daß das sequentielle Vergleichsprogramm (mit seinem begrenzten Speicherplatz) diese Sicherheit auch noch erreichen kann.

Beispiel: Für eine maximale parallele Rechenzeit von 480 Sekunden ergibt sich für die Speedupmessung von 159 Prozessoren die folgende Menge von Stellungs-Sicherheits-Tupeln $(S, T = (\text{Tiefe } t, \text{Conspiracy Number } c))$:
$\mathcal{P} = \{ (1,(10,2)), (2,(15,5)), (3,(13,2)), (4,(11,2)), (5,(15,5)), (6,(12,2)), (7,(13,2)),$
$(8,(13,2)), (9,(12,2)), (10,(13,2)), (11,(12,2)), (12,(12,2)), (13,(10,2)), (14,(14,2)),$
$(15,(11,2)), (16,(9,2)), (17,(14,2)), (18,(21,5)), (19,(11,2)), (20,(10,2)), (21,(14,2)),$
$(22,(11,2)), (23,(12,5)), (24,(15,2)), (25,(16,2)), (26,(10,2)), (27,(13,2)), (28,(10,2)),$
$(29,(10,2)), (30,(12,2))\}$. Auf z.B. der ersten Stellung des BT2630-Tests werden also die Rechenzeiten einer Suche mit Tiefe 8 und Conspiracy Number 2 ermittelt. Rechenzeitschranken für parallele Rechenzeiten waren 20, 45, 90, 180, 240, 480, 720, 900 und 10000 Sekunden.
Bei einer vorgegebenen Rechenzeitschranke (z.B. 480 Sekunden) ergibt der durchschnittliche Speedup aller 30 Stellungen des BT-Tests einen *Meßpunkt*. Verschiedene Meßpunkte, die durch verschiedene Beschränkungen der parallelen Rechenzeit erzeugt werden, werden durch lineare Interpolation miteinander verbunden.

Es gibt aber noch ein weiteres meßtechnisches Problem: Der Speicher der Prozessoren des HPCLine-Rechners ist auf 256 MByte begrenzt, wobei davon ca. 100 MByte vom Betriebssystem verbraucht werden. Um sequentielle Daten aufzunehmen, ist diese geringe Speichermenge nicht geeignet. Wir haben deshalb die sequentiellen Daten auf einer SUN Workstation mit 2 GByte Speicher durchgeführt, dem Rechner Quattro. Eine Umrechnung ist nicht trivial. Durch verschieden schnelle Hauptspeicher, verschiedene Cache-Strategien und letztendlich durch unterschiedliche Befehlssätze der Prozessoren ist die Geschwindigkeitsdifferenz von einem System zum anderen von der Anwendung abhängig. Der von uns gemessene Umrechnungsfaktor beträgt 1.31. Wie man in Abbildung 3.15 sieht, ist dieser Faktor aber selbst innerhalb unserer Anwendung Schwankungen unterworfen. Um das Verhältnis der Rechengeschwindigkeiten der unterschiedlichen Plattformen abzuschätzen, haben wir eine Vergleichsmessung auf dem BT2630-Test durchgeführt. Für lange Rechnungen sind wir zu dem Schluß gekommen, daß ein Umrechnungsfaktor von 1.31 angemessen ist. Abbildung 3.15 zeigt das Verhältnis der Ge-

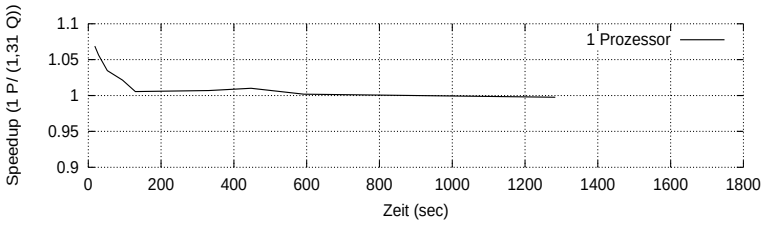


Abb. 3.15: HPCLine vs. Quattro

schwindigkeit eines Pentiumprozessors zum 1,31-fachen der Geschwindigkeit des Rechners Quattro. Zu Beginn einer Rechnung ist ein Pentiumprozessor des HPC-Line Workstationclusters mehr als 1,31 mal schneller, als der Rechner Quattro (ca. 7%). Zwischen 600 und 1200 Sekunden Berechnungsdauer ist der Umrechnungsfaktor 1,31 passend, danach scheint sich das Geschwindigkeitsverhältnis noch weiter zugunsten der SUN-Workstation zu entwickeln. Aufgrund des begrenzten Speichers konnten wir die Entwicklung nicht weiter beobachten. Wir gehen davon aus, daß sich das Geschwindigkeitsverhältnis allenfalls noch in einem kleinen Bereich ändert ($<5\%$). Falls die Leistung des Pentiumprozessors wider Erwarten weiter relativ zu der des Sparcprozessors fallen sollte, bedeutete das lediglich, daß die von uns gemessenen Speedups des parallelen Verfahrens besser sind als im folgenden dargestellt.

3.5.4 Speedupmessungen für senderinitiierte Cc2s

In diesem Kapitel betrachten wir das Verhalten unseres verteilten Algorithmus auf verschiedenen Maschinen mit bis zu 159 Prozessoren. Alle Ergebnisse wurden auf den Stellungen des Bednorz-Tönnissen Tests BT2630 ermittelt. Das Programm benutzt dabei alle bisher beschriebenen Methoden und Heuristiken zur Beschleunigung der Suche (vgl. Abschnitte 2.1 bis 2.3 und 3.4.2). Wir betrachten zuerst Messungen auf dem HPCLine Rechner, bei denen wir die MPI-Version der Firma SCALI benutzen. Diese MPI-Version verwendet die in Abschnitt 3.1 beschriebene SCI-Netzwerktechnologie. Danach präsentieren wir noch einmal Messungen vom HPCLine-Rechner, bei denen dann aber die Standardbibliothek MPICH für Ethernettechnologie eingesetzt wird. Zum Abschluß präsentieren wir Speedupergebnisse des CC48 Rechners.

System: HPCLine mit ScaMPI

In Abbildung 3.16 sieht man, wie sich die Speedups für unterschiedliche Prozessorzahlen bei wachsenden Rechenzeiten entwickeln. Es fällt auf, daß die Speedups für alle Prozessorzahlen in den ersten 100 Sekunden ansteigen und danach mehr oder weniger kon-

stant bleiben. Wie man in den Abbildungen 3.18 und 3.17 erkennen kann, steigt in den

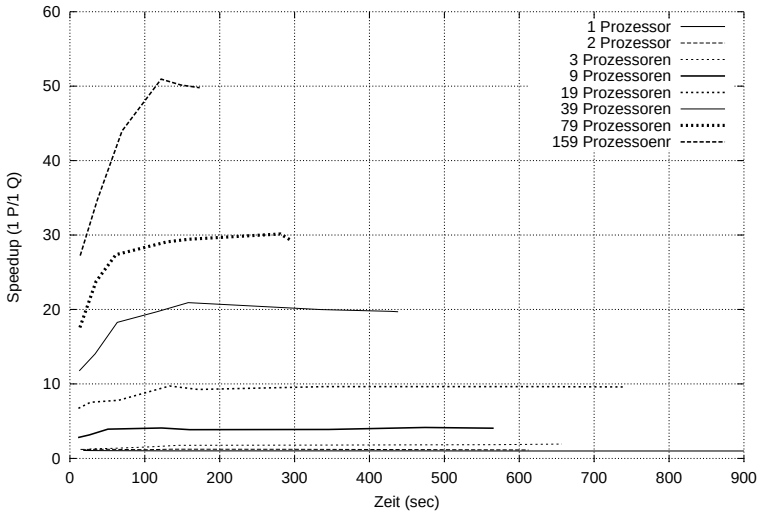


Abb. 3.16: Speedups auf dem HPCLine System

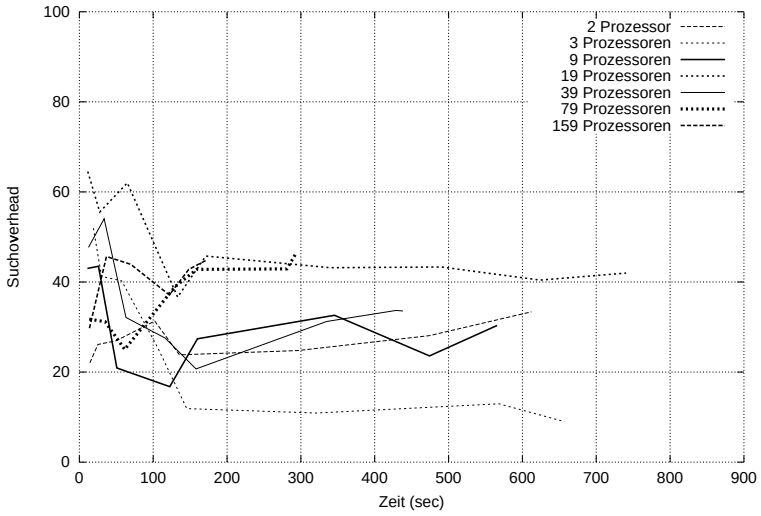


Abb. 3.17: Searchoverhead auf dem HPCLine System

ersten 100 Sekunden die Auslastung an, während der Suchoverhead gleichzeitig fällt. Der Grund dafür ist, daß die zu Beginn einer Berechnung zu lösenden Teilprobleme zu klein sind, um alle Prozessoren mit sinnvoller Arbeit zu versorgen. So werden z.B. von DetermineMove(. . .) zuerst einmal alle Züge an der Wurzel erzeugt und bewertet. Das Prozessornetzwerk ist in dieser Zeit schlecht ausgelastet.

Erst wenn der zu untersuchende Spielbaum eine gewisse Größe erreicht hat, finden alle Prozessoren sinnvolle Arbeit. Damit während dieser ersten kritischen Phase nicht zu viele Prozessoren arbeitslos sind, werden auch Teilexpansionen von Knoten parallel (Abschnitt 3.4.2) bearbeitet. Die Vorhersagen, welche dieser Spielbaumexpansionen tatsächlich gebraucht werden, sind seltener erfolgreich, als dies bei der Aufteilung von Targets an inneren Knoten der Fall ist. Gleichzeitig bewertete Knoten bei einer Spielbaumerweiterung führen deshalb zu unvorsichtigem Einsatz von Parallelität und damit zu mehr Suchoverhead.

Für Prozessorzahlen bis 39 werden (ab ca. 150 Sekunden paralleler Rechenzeit) Effizienzen von 50% (Höchstwert bei 39 Prozessoren: 53%), für 79 Prozessoren eine Effizienz von 38%, und für 159 Prozessoren eine Effizienz von 32% erreicht.

Die Suchoverheadmessungen (vgl. Abb. 3.17) geben keinen endgültigen Aufschluß darüber, ob eine steigende Prozessorzahl auch zu steigendem Suchoverhead führt. Zwar entspräche das unseren Erwartungen, und 2,3 und 9 Prozessoren benötigen auch sichtbar weniger Suchoverhead als 79 oder 159 Prozessoren, aber auch 19 Prozessoren untersuchen mit über 40% Suchoverhead sehr viele Knoten. Erfreulich ist, daß sich der Suchoverhead im Laufe der Zeit nicht erhöht, sondern nahezu gleich bleibt. Der Grund dafür,

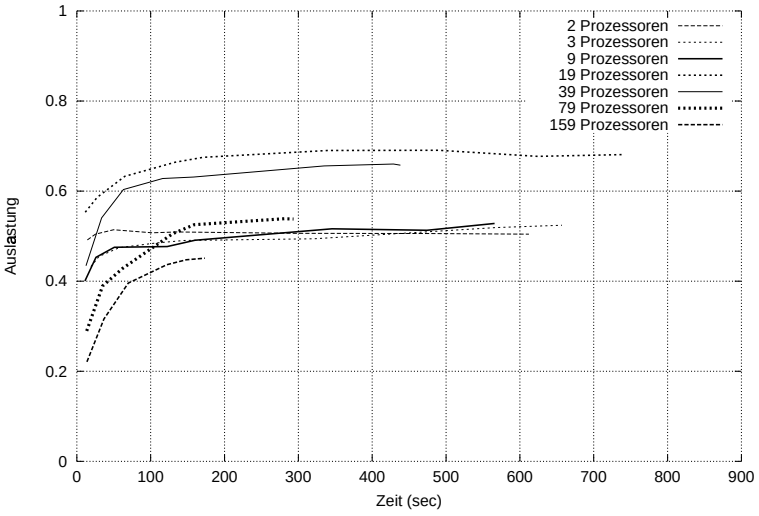


Abb. 3.18: Auslastung auf dem HPCLine System

daß sich die Effizienz mit zunehmender Prozessorzahl verschlechtert, ist eher in der begrenzten Auslastung zu sehen (vgl. Abb. 3.18). Die entstehenden Suchbäume sind sehr unregelmäßig und schmal. Deshalb gibt es immer wieder mal Zeitabschnitte, in denen kleine Teilprobleme erst zu Ende gerechnet werden müssen, bevor deren Ergebnisse neue

kleine Teilprobleme initiieren, bei denen der Cc2s-Algorithmus auch wieder nur wenige Prozessoren bei der Auswertung zum Einsatz bringen kann (vgl. [LR96]). Wenn der Cc2s-Algorithmus ein Teilproblem bearbeitet, bei dem er maximal (sagen wir) 5 Prozessoren einsetzen kann, wirkt sich das in einem großen Prozessornetzwerk mit 159 Prozessoren natürlich wesentlich stärker aus als in einem kleinen mit nur 9 Prozessoren. Der hier vorgestellte Algorithmus arbeitet in diesen Phasen nicht feingranular genug, um die vorhandene Last verteilen zu können.

Die Maschine, auf der P.ConNerS läuft, begrenzt ebenfalls die maximal mögliche Auslastung: Wenn die Ringe, auf denen die Nachrichten wandern müssen, saturiert werden, führt das zu Verzögerungen von Nachrichten. Wir werden im nächsten Abschnitt sehen, daß langsamerer Nachrichtenaustausch die Auslastung des Systems deutlich verschlechtert.

Kaum relevant sind andere Einflüsse, wie z.B. die Schedulingzeiten, in denen Tasks ausgetauscht werden. Die sind bei den Messungen schon in der Auslastung enthalten.

Im Bereich ab 100 Sekunden paralleler Laufzeit befinden sich die Lastkurven (Abbildung 3.18) zwischen 45 und 70 Prozent. Daß die Auslastungen bei 19 und 39 Prozessoren mit Abstand besser sind als bei den übrigen Prozessoreanzahlen, liegt unseres Erachtens nach nur daran, daß uns zu Testzwecken meistens Partitionen mit bis zu 39 Prozessoren zur Verfügung standen. Ohne es zu beabsichtigen haben wir das Gesamtsystem auf diese Partitionsgrößen optimiert.

System: HPCLine mit MPICH

Wir zeigen nun, wie wichtig und für eine Parallelisierung und für unseren verteilten Cc2s-Algorithmus entscheidend die Kommunikationsgeschwindigkeit eines Parallelrechners ist. Wir messen dazu das Verhalten unseres Schachprogramms auf bis zu 79 Prozessoren, verzichten aber auf den Einsatz der SCI unterstützten MPI-Library der Firma SCALI und benutzen stattdessen eine Fastethernet-Verbindung und die MPICH-Standardlibrary.

Interessanterweise erhalten wir auch hier nach ca. 150 bis 200 Sekunden den maximalen Speedup (Abb. 3.19). Allerdings ist er wesentlich niedriger als unter dem SCI-Verbindungsnetzwerk. Mit 79 Prozessoren kommt der Algorithmus nicht über einen Speedup von 12 hinaus. Erreicht das Programm mit 9 Prozessoren noch eine Effizienz von ca. 40% (nach 200 Sekunden paralleler Laufzeit), liegt diese bei 79 Prozessoren nur noch bei 15% (nach 200 Sekunden paralleler Laufzeit). Aufgrund unserer Vorüberlegungen in Abschnitt 3.3.1 hatten wir erwartet, den Grund für die niedrigen Effizienzen sowohl in niedriger Auslastung als auch in hohem Suchoverhead zu finden.

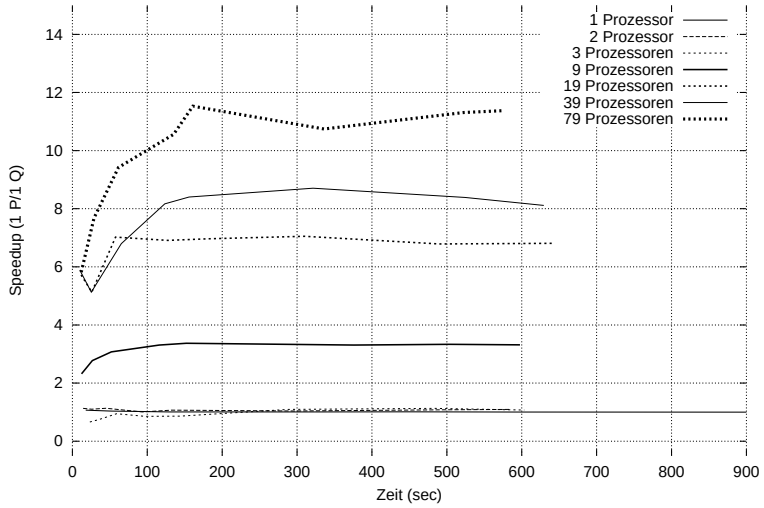


Abb. 3.19: Speedups auf dem HPCLine System unter MPICH

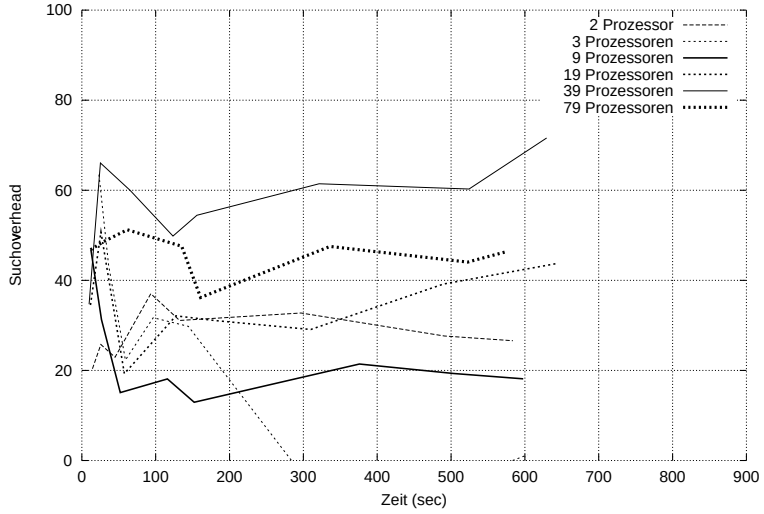


Abb. 3.20: Searchoverhead auf dem HPCLine System unter MPICH

Interessanterweise ist der Suchoverhead nahezu gleich dem Suchoverhead der SCI-Version (Abb. 3.20). (Die einzige Ausnahme bildet hier der Lauf mit 39 Prozessoren, der teilweise über 70% Suchoverhead erzeugte.) Der eigentliche Grund für die schlechten Speedups bei der Verwendung der MPICH-Programmbibliothek liegt wieder in der niedrigen Auslastung (Abb. 3.21) des Prozessornetzwerkes. Schon bei kleinen Prozessorzahlen (z.B. 3 und 9) liegt sie unter 40%. Bei 79 Prozessoren sinkt sie sogar auf 20% ab. Da aber die Anzahl der vorhandenen Teilprobleme in etwa gleich hoch ist wie bei der SCI gestützten

Version (da der Suchoverhead weitestgehend konstant bleibt und er auch nicht von der Kommunikationsgeschwindigkeit abzuhängen scheint, gehen wir davon aus, daß die Anzahl der insgesamt zu erzeugenden Teilprobleme in erster Linie von der Ausgangsstellung und der geforderten Ergebnissicherheit an der Wurzel abhängt), kann die Granularität der Parallelisierung hier nicht als Grund für die niedrige Auslastung angeführt werden.

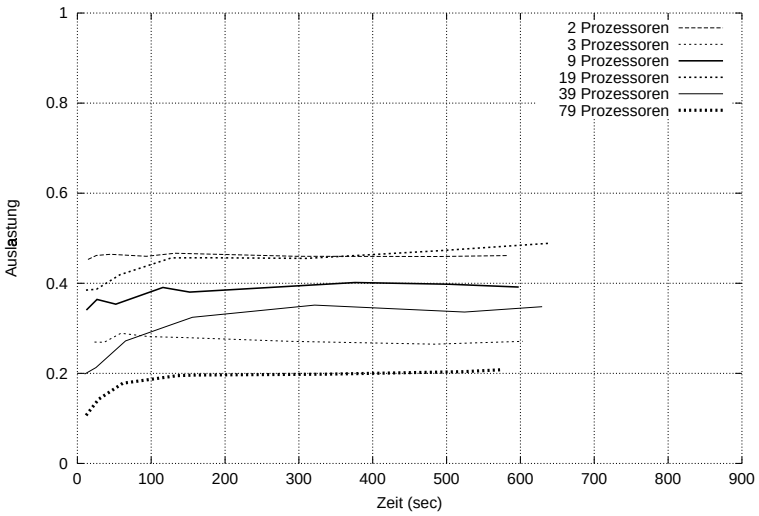


Abb. 3.21: Auslastung auf dem HPCLine System unter MPICH

Beobachtung 3.5-5
Die Effizienz des parallelen Cc2s-Algorithmus hängt in erster Linie von der erreichten Auslastung des Prozessornetzwerkes ab. Beim Vergleich der SCI- und Fastethernet-Netzwerke stellt man fest, daß die erreichte Auslastung entscheidend von der zugrunde liegenden Hardware abhängt. □

CC48

Die Meßergebnisse, die wir auf dem CC-48 Rechner erhalten haben, sind sämtlichst aus dem Jahr 1997 und somit wesentlich älter als die Ergebnisse des HPCLine Rechners. Leider existiert der CC-48 nicht mehr, so daß wir auch keine neueren oder genaueren Ergebnisse vorzeigen können. Aufgrund seiner Topologie und dem damit erreichten Verhältnis von Kommunikationsgeschwindigkeit zu Rechengeschwindigkeit der Prozessoren konnten dort, auch ohne die Parallelität der letzten Ebene zu nutzen (Abschnitt 3.4.2), Effizienzen von ca. 50% erreicht werden.

#	Speedup	Suchoverhead
5	3.0	24%
9	5.9	41%
19	12.3	33%
39	19.8	48%

Tab. 3.3: Ergebnisse auf dem CC48 Rechner

3.5.5 Streuungsmessungen auf der HPCLine

Die im vorigen Kapitel vorgestellten Ergebnisse beruhen auf jeweils einem festen Testlauf. Wir untersuchen nun das Streuungsverhalten von Speedup Testläufen mit 39 Prozessoren unter Verwendung des SCI-Netzwerkes.

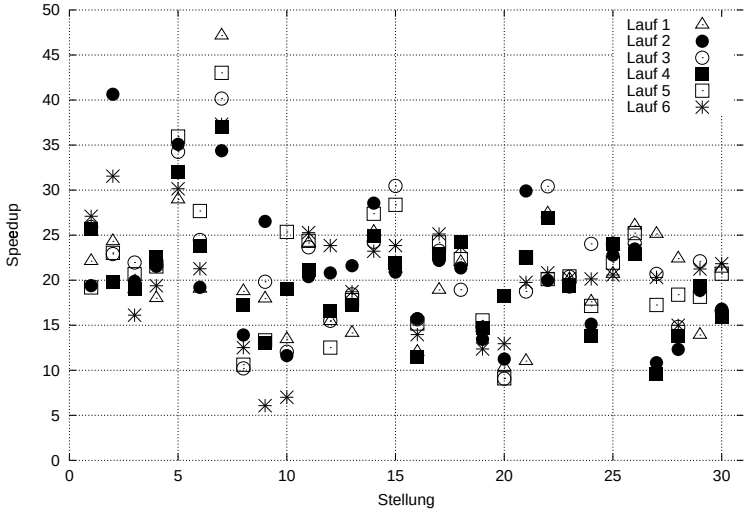


Abb. 3.22: Streuung bei 39 Prozessoren

Der in dieser Arbeit vorgestellte Cc2s-Algorithmus arbeitet in hohem Maße nichtdeterministisch, wenn er auf mehr als einem Prozessor eingesetzt wird. Der Nichtdeterminismus entsteht dadurch, daß wir nicht spezifiziert haben, zu welchen Zeitpunkten und in welcher Reihenfolge Nachrichten bei einem Prozessor eintreffen. In der Tat ist das genau die Situation, die wir auf einem lose gekoppelten Prozessornetzwerk im allgemeinen vorfinden. Asynchrone Kommunikationsmechanismen, z.B. unter MPI, zeichnen sich dadurch aus, daß zwar garantiert wird, daß eine Nachricht, die von einem Prozessor abgeschickt wurde, sein Ziel irgendwann erreicht, aber es werden keine Angaben gemacht, zu welchem Zeitpunkt genau die Nachricht ankommt. Kleinste Veränderungen der Luftfeuchtigkeit, der Raumtemperatur, der Ausgangskonfiguration etc. sorgen für erhebliche Veränderungen.

Um Streuungen in der parallelen Laufzeit zu untersuchen, haben wir beispielhaft für 39 Prozessoren jeweils sechs Testläufe durchgeführt. Jeder einzelne Testlauf wurde wie im vorigen Abschnitt beschrieben erstellt. Die Speedupergebnisse jeder einzelnen Stellung aller sechs Testläufe sind in Abbildung 3.22 dargestellt. Dabei ergeben sich teilweise erhebliche Streuungen bei einzelnen Teststellungen. Die stärksten Streuungen ergeben sich bei den Stellungen BT2 und BT9.

Bei Stellung BT2 ergibt sich ein Mittelwert von

$$\frac{24,3 + 40,65 + 22,9 + 19,8 + 23,03 + 31,11}{6} = 26,97.$$

Die maximale Abweichung vom Mittelwert in Prozent, bezogen auf den Mittelwert beträgt somit

$$\frac{40,65 - 26,97}{26,97} \cdot 100\% = 50,7\%.$$

Bei BT9 ist der Mittelwert 16,43 und die prozentuale Abweichung sogar 63%.

Bei anderen Stellungen, wie z.B. BT23 beobachtet man wesentlich geringere Abweichungen. Dort beträgt der Mittelwert

$$\frac{20,11 + 19,24 + 20,38 + 19,35 + 20,42 + 20,19}{6} = 19,95.$$

und die maximale Abweichung vom Mittelwert in Prozent, bezogen auf den Mittelwert

$$\frac{19,95 - 0,71}{19,95} = 3,6\%.$$

Die Schwankungen des Speedups auf der gesamten Reihe sind gering. Der durchschnittliche Speedup schwankt zwischen 18,92 und 19,95. Das aritmethische Mittel ergibt

$$\frac{18,92 + 19,69 + 19,95 + 19,28 + 19,58 + 19,45}{6}.$$

Damit ist die maximale Abweichung des durchschnittlichen Speedups vom Mittelwert lediglich 2,87%.

Beobachtung: Die Streuungsmessungen zeigen, daß die ausgewählte Testreihe BT2630, bestehend aus 30 Einzelstellungen, ausreichend viele Stellungen enthält, so daß die einzelnen, auftretenden Schwankungen der gemessenen Speedups den durchschnittlichen Speedup über alle 30 Stellungen nicht zu stark beeinflussen.

Für eine Beurteilung verschiedener Programmversionen genügt es diese auf einen einzelnen Testlauf über 30 Stellungen zu stützen.

Kapitel 4

Spielbaumsuche mit Fehlern

Spielbaumsuche greift, wie schon im ersten Kapitel erläutert, die Probleme auf, die entstehen, wenn wir Computer z.B. dazu bringen möchten, Zweipersonen-Nullsummenspiele wie Schach, Dame, Othello u.s.w. zu spielen. Der bisher größte Erfolg von Spielbaumsuche war der Sieg von der Schachmaschine 'Deep Blue' gegen den Weltmeister und besten menschlichen Schachspieler der Welt, Garry Kasparov.

Trotz der großen Popularität von Computerschach, das geradezu als Inbegriff von künstlicher Intelligenz angesehen wird, gibt es bislang kein allgemein anerkanntes Modell, das diese Erfolge erklären kann oder in dem man gar in der Lage ist, nützliche Anleitungen herzuleiten, wie man (selektive) Spielbaumsuche einsetzen kann.

In diesem letzten Kapitel stellen wir ein kombinatorisches Modell vor, das Fehler einer heuristischen Bewertungsfunktion mit Hilfe von Würfelexperimenten modelliert. Unter der Annahme, daß man eine Bewertungsfunktion zur Verfügung hat, deren Fehlerwahrscheinlichkeit zwar größer als Null, aber doch beliebig nahe an Null ist, wird das Ergebnis sein, daß eine Spielbaumsuche auf einem fest vorgegebenen Spielbaum G genau dann sinnvoll ist, wenn es in dem zu untersuchenden Spielbaum G wenigstens zwei blattdisjunkte Strategien gibt, die den Wert an der Wurzel begründen. Außerdem werden wir für eine solch gute Bewertungsfunktion zeigen, daß in einem beliebigen, aber fest gewählten Spielbaum G die Anzahl blattdisjunkter Strategien, die den Wert der Wurzel begründen, qualitativ die Güte eines heuristischen Minimaxwertes bestimmt. Wie klein die Fehlerwahrscheinlichkeit der Bewertungsfunktion gewählt werden muß, damit unsere Aussagen gelten, hängt vom jeweiligen Spielbaum ab.

Alles das führt zu einer zumindest intuitiven Erklärung dafür, weshalb der Einsatz von Conspiracy Number Search sinnvoll ist.

4.1 Einleitung

4.1.1 Das Vorgehen der Anwender

Wenn ein Spielbaum so groß ist, daß er nicht vollständig durchsucht werden kann und es somit nicht möglich ist, einen tatsächlich besten Zug zu bestimmen, benutzen Computer die folgende Methode, um zu Entscheidungen zu gelangen: Zunächst wird ein Teilbaum des Gesamtspielbaums herausgesucht. Dies kann ein beliebiger Teilbaum sein, der die Wurzelstellung des Gesamtspielbaums als Wurzel hat. Z.B. kann es sich um einen Spielbaum handeln, bei dem alle Blätter eine bestimmte Entfernung zur Wurzel haben. Wir haben diesen ausgewählten Teilbaum bereits als *Hülle* bezeichnet. Dann weist ein Suchalgorithmus jedem Blatt der Hülle einen heuristischen Wert zu und rechnet den Minimaxwert dieser Zahlen zur Wurzel hoch. Meistens wird zu diesem Zweck der sog. $\alpha\beta$-Algorithmus [KM75] herangezogen. Was Fehlerhäufigkeiten an der Wurzelstellung angeht, macht es allerdings keinen Unterschied, ob der $\alpha\beta$-Algorithmus oder ein einfacher Minimax-Algorithmus verwendet wird. Das Ergebnis ist immer das gleiche, nur im Aufwand, um das Ergebnis auszurechnen, unterscheiden sich die Algorithmen erheblich.
Folgende Beobachtung kann man als gesichert ansehen:

Beobachtung 4.1-6
Das Minimax-Auswerten von Spielbäumen, bei dem Spielbaumblätter heuristisch bewertet werden, führt in vielen Spielen (insbesondere beim Schachspiel) zu besseren Abschätzungen an der Wurzel des Spielbaums als eine direkte heuristische Bewertung der Wurzel.
□

Es stellt sich nun die Frage, ob dieser Effekt naturgegeben ist oder ob möglicherweise bestimmte strukturelle Gegebenheiten der Spielbäume zu dem beobachteten Effekt führen.

4.1.2 Fehlermodelle anderer Autoren

Pearl [Pea84] untersuchte Spielbäume, bei denen den Blättern die Werte GEWINN und VERLUST zufällig zugeordnet werden. Das Ergebnis ist, daß in so einem Modell der Minimaxwert nicht von der inneren Baumstruktur, sondern nur von der Wahrscheinlichkeit abhängt, mit der ein Blatt ein GEWINN-Blatt ist.

Das meistbenutzte Modell zur Fehleranalyse [Pea83], [Nau79], [Nau82], [BG82] geht davon aus, daß Blätter eines Spielbaums fehlerhaft mit einer Wahrscheinlichkeit p' bewertet werden. Fehlbewertungen geschehen dabei voneinander unabhängig.

Schrüfer [Sch86] schlug ein Modell vor, in dem er Klassen von Spielbäumen top-down konstruiert. Für bestimmte Spielbaumklassen konnte er nachweisen, daß Spielbaumsuche bis zu einer möglichst großen, festen Tiefe nützlich ist. Außerdem konnte er für ein spezielles stochastisches Spielbaummodell (das sogenannte S-Baummodell) angeben [Sch88], unter welchen Bedingungen ein Spiel *vertiefungsfreundlich* ist. Das Modell wird durch folgende Parameter bestimmt:

- b aus $\mathbb{N}$, mit $b \geq 2$ ist der Branchingfaktor der Spielbäume.
- Echte Werte sind aus $\{0, 1\}$ und gehorchen dem Minimaxprinzip. Man kann dabei zwei Knotentypen unterscheiden: Nennen wir sie ALL-Knoten (das sind die, an denen wegen des Minimaxprinzips alle Nachfolger den Wert des Vaters annehmen müssen) und CUT-Knoten.
- $s = p_1, \ldots, p_b$ sind Wahrscheinlichkeiten dafür, dass $1, 2$, oder b viele Nachfolger den Wert eines CUT-Knotens annehmen.
- e^+ ist die Wahrscheinlichkeit dafür, daß ein ALL-Knoten falsch eingeschätzt wird.
- e^- die Wahrscheinlichkeit dafür, daß ein CUT-Knoten falsch eingeschätzt wird.

Sei nun S ein Spiel $(b, p_1, .., p_b, e^-, e^+)$, und bezeichnen $e^+(t)$ und $e^-(t)$ die Wahrscheinlichkeiten für falsche Einschätzungen von auf Tiefe t eingeschränkten Teilspielen. S selber läßt sich als Wahrscheinlichkeitsverteilung über alle möglichen im Modell enthaltenen Spielbäume mit Branchingfaktor b interpretieren.
Ein Spiel S heißt *vertiefungsfreundlich*, wenn es x^+, x^- aus dem Intervall $[0, 1]$ gibt, so daß für alle t aus $\mathbb{N}$ gilt:
$e^+(t) < x^+$ und $e^-(t) < x^- \Rightarrow e^+(t+2) < e^+(t)$ und $e^-(t+2) < e^-(t)$. Der Begriff *vertiefungsfeindlich* ist analog definiert.
Als Ergebnis bekommt Schrüfer heraus:
a) $b \cdot p_1 < 1 \Rightarrow S$ ist vertiefungsfreundlich und
b) $b \cdot p_1 > 1 \Rightarrow S$ ist vertiefungsfeindlich.
Für dieses Modell wurden weitere interessante Ergebnisse präsentiert [Alt88] [Alt90a].

Beal benutzt ebenfalls ein Grundmodell, bei dem Blattwerte mit einer gewissen Wahrscheinlichkeit p' falsch eingeschätzt werden. Er schränkt sich aber auf Spielbäume ein, bei denen sich die Blattwerte des Spielbaums clustern [Bea80], da er der Meinung ist, daß das Ausgangsmodell zu einfach sei, um tiefgreifende Aussagen ausrechnen zu können [Bea82] [Bea99].

Wir erarbeiten in dieser Arbeit zwei sehr einfache Modelle, von denen das zweite ebenfalls davon ausgeht, daß Bewertungsfehler an Blättern eines Spielbaums zufällig gemacht werden und daß diese Bewertungsfehler voneinander unabhängig sind. Es liefert uns sehr starke Aussagen, die für alle Spielbäume (also für jeden einzelnen) und nicht nur für eine eingeschränkte Klasse von Spielbäumen oder im Durchschnitt über alle Spielbäume gelten.

Wegen der PSPACE-Schwere einiger Zweipersonen-Nullsummenspiele können wir nicht erwarten, daß es uns gelingt, eine einfache Formel zu finden, die uns eine Aussage darüber gibt, ob ein heuristischer Minimaxwert mit dem echten übereinstimmt, zumindest nicht für alle Spiele, alle Spielsituationen und alle Spielbäume, die zu diesen Spielsituationen gehören. Solch eine Formel würde uns erlauben, den echten Wert der Wurzel schnell zu berechnen.

Deshalb versuchen wir, die Beschreibung von Fehlern und deren Fortpflanzung in Spielbäumen zu vereinfachen. Wir unterscheiden zwischen der Fehleranzahl (Fehlerrate) und den Fehlerpositionen. Auf diese Weise kann man die Fehler z.B. in folgende drei Klassen unterteilen: Eine bestimmte Menge von Fehlern kann bösartig, gutartig oder auf neutrale Art positioniert werden. 'Neutrale Art' soll hier nicht heißen, daß die Fehler so positioniert werden, daß sie keine Auswirkung auf das Wurzelergebnis haben (das wäre eine gutartige Positionierung), sondern mit 'neutral' meinen wir, ohne es genauer spezifizieren zu wollen, so etwas wie durchschnittlich oder zufällig. Da die Analyse der ersten beiden Fälle trivial ist (vgl. Modell 1) und im allgemeinen auch eine neutrale Fehlerpositionierung am realistischsten ist, konzentrieren wir uns auf den letzten Fall (vgl. Modell 2). Diese Art der Positionierung modellieren wir durch zufällige Fehler, bei vorgegebener Fehlerwahrscheinlichkeit.
Wir stellen in den folgenden Abschnitten unsere Modelle vor. Begriffe wie 'Spielbaum', 'Strategie', 'Minimax' u.s.w. wurden bereits in Kapitel 1.8 erläutert.

4.2 Spielbaumsuche über Bewertungen mit Fehlern

Modell 1: Es sei ein beliebiger, endlicher Spielbaum $G = ((V, E), f)$ vorgegeben. Jeder seiner Knoten habe entweder den Wert 0 oder 1 (durch f bestimmt), und diese 'echten' Werte gehorchen dem Minimax-Prinzip. Außerdem soll es eine Bewertungsfunktion h geben, die die Knoten des Spielbaums auf sogenannte 'heuristische' Werte abbildet. h ist also wie f eine Funktion von Knoten in die Menge der möglichen Werte: $h : V \to \{0, 1\}$. Wenn $f(v)$ ungleich $h(v)$ ist (echter und heuristischer Wert nicht übereinstimmen), sprechen wir davon, daß h am Knoten v einen Bewertungsfehler macht.

Auch wenn es inkonsequent erscheinen mag, daß wir die Funktion f als Teil der Spielbaumdefinitionen schreiben und h nicht, erschien uns eine Umdefinition des Spielbaumbegriffs nicht sinnvoll. Die Funktion h ist lediglich eine Hilfskonstruktion, die es uns ermöglichen soll, mit den einfachen formalen Mitteln von kombinatorischen Modellen auszukommen.
Weiterhin ist uns klar, daß man in der Praxis eher mit heuristischen Bewertungsfunktionen arbeiten wird, die einen größeren Wertebereich als $\{0, 1\}$ haben. Zum einen erleichtert diese Einschränkung aber unsere Analysen, zum anderen reicht diese Wertemenge aus, um die von uns als wichtig erachteten Effekte zu beschreiben.

Die heuristischen Werte genügen nicht unbedingt dem Minimaxprinzip. Man kann allerdings einen heuristischen Minimaxwert für die Wurzel eines Spielbaums ausrechnen, indem man die heuristischen Blattwerte nach dem Minimaxprinzip zur Wurzel hochpropagiert. Wir benutzen bei der Analyse das Minimaxprinzip, weil es dasjenige Prinzip ist, das sich in der Praxis durchgesetzt hat (s.o). Man kann sich aber auch andere Verfahren zur Wertbildung innerer Knoten vorstellen [Alt90b].

Rufen wir uns noch einmal den Begriff der blattdisjunkten Strategien ins Gedächtnis:

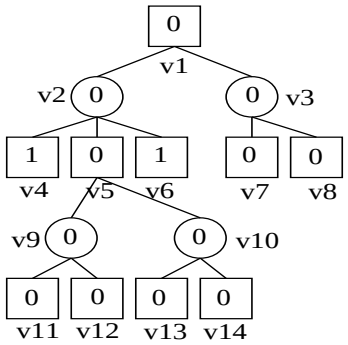


Abb. 4.1: Spielbaum G

Um in dem hier gegebenen Beispiel (Abb. 4.1) zu zeigen, daß der Wert der Wurzel von G den Wert 0 hat, brauchen wir nur den Teilbaum auszuwerten, der aus den Knoten $S_1 := \{v1, v2, v3, v5, v8, v9, v10, v12, v14\}$ besteht. Wir nennen diesen Teilbaum, der den Wert 0 als obere Schranke beweist, eine *belegende (Beweis)-Strategie*. Die Knoten $S_2 := \{v1,v2,v3,v5,v8,v9,v10,v11,v14\}$ bilden ebenfalls eine Beweisstrategie für den Wert 0. Im Gegensatz zu $S_3 := \{v1,v2,v3,v5,v7,v9,v10,v11,v13\}$ ist sie jedoch nicht *blattdisjunkt* zu S_1. S_1 und S_3 sind blattdisjunkte Strategien, die beide den Wert 0 an der Wurzel beweisen.

Folgende zwei Aussagen sind einfach einzusehen:

Wenn G n blattdisjunkte Strategien enthält, die alle den Wurzelwert belegen, darf h mindestens $n-1$ viele Blätter falsch bewerten, ohne daß der heuristische Minimaxwert der Wurzel von G falsch wird.

Wenn G genau n solcher blattdisjunkte Strategien enthält und h n Fehler macht, kann man durch eine bösartige Positionierung dieser Fehler erreichen, daß auch der Wurzelwert falsch wird.

4.3 Spielbaumsuche über Bewertungen mit zufälligen Fehlern

Im Normalfall wird nicht gerade unser Gegner die Fehler unserer Bewertungsfunktion positionieren! Insofern ist die obige Aussage unbefriedigend. Man ist doch eher daran interessiert, was passiert, wenn die Fehler irgendwie, neutral verteilt werden. Wenn man also z.B. zusätzlich annimmt, G habe m viele Blätter, und h mache k viele Fehler, möchte man wissen, in wievielen der $\begin{pmatrix} m \\ k \end{pmatrix}$ möglichen Fälle, k Fehler auf m Blätter zu verteilen, der heuristische Minimaxwert richtig ist und mit dem echten Wert übereinstimmt.

Diese Fragestellung scheint für alle möglichen Spielbäume nur schwerlich analysierbar zu sein.

Man kann die Fragestellung aber leicht abwandeln: Was läßt sich aussagen, wenn man davon ausgeht, daß G m viele Blätter besitzt und h ungefähr $(1-p)\cdot m$ viele Fehler macht, wobei $p \in (0,1)$ ist? Diese Frage führt direkt zu Modell 2:

Modell 2: Es sei ein beliebiger, endlicher Spielbaum $G = ((V,E),f)$ vorgegeben. Jeder seiner Knoten habe entweder den Wert 0 oder 1, und diese 'echten' Werte gehorchen dem Minimax-Prinzip. Wir führen Münzwürfe an den Blättern durch, so daß ein Blatt mit Wahrscheinlichkeit p seinen Wert behält und mit Wahrscheinlichkeit $1-p$ den komplementären Wert zugewiesen bekommt. Der Einfachheit halber gehen wir davon aus, daß p für alle Blätter gleich ist. An den inneren Knoten von G bilden wir die Minimaxwerte der verfälschten 'heuristischen' Werte. Nachdem alle heuristischen Blattwerte ausgewürfelt sind, beschreibe eine Funktion h die Abbildung der Knoten in die heuristischen Werte. Die zu untersuchende Fragestellung ist, mit welcher Wahrscheinlichkeit der echte und der heuristische Minimaxwert der Wurzel von G gleich sind.

Wir werden zeigen, daß der Begriff der 'blattdisjunkten Strategien' auch in diesem Modell eine ganz zentrale Bedeutung besitzt. Unter der Annahme, daß die Wahrscheinlichkeit dafür, daß h einen fehlerhaften Wert liefert, nahe genug bei Null ist, werden wir beweisen, daß die Anzahl der blattdisjunkten Beweisstrategien, die alle den Wurzelwert beweisen,

bestimmt, wie gut der Wurzelwert durch einen heuristischen Minimaxwert approximiert wird. Wenn nicht wenigsten 2 solcher blattdisjunkten Strategien vorhanden sind, führt (bei Verwendung einer fast perfekten Bewertungsfunktion h) eine Minimaxauswertung des gegebenen Spielbaums zu einer schlechteren Lösung als eine direkte heuristische Bewertung der Wurzel.

Man beachte, daß unsere Ergebnisse für alle(!) Spielbäume gelten. Sie sind weder eingeschränkt auf Spielbäume mit fester Tiefe noch auf Spielbäume, bei denen sich die Werte in irgendeiner Weise clustern. Darüberhinaus werden lediglich die Fehler an den Blättern als zufällig angenommen, nicht die Spielbäume selber.

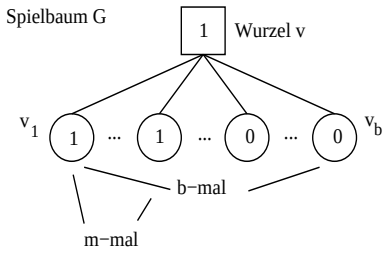
4.3.1 Analyse

Sei ein beliebiger, aber endlicher Spielbaum G in Modell 2 gegeben. Im folgenden bezeichne ϵ die Wurzel von G. Wie groß ist die Wahrscheinlichkeit, daß h(ϵ) = f(ϵ) in Abhängigkeit von der Nichtfehlerwahrscheinlichkeit p an den Blättern von G ist, wenn wir davon ausgehen, daß einzelne Fehler vollständig voneinander unabhängig gemacht werden? (Wir können uns z.B. vorstellen, daß die Fehler durch Münzwürfe erzeugt werden.)

Wegen der Baumstruktur von G sind dann Nichtfehlerwahrscheinlichkeiten von blattdisjunkten Teilbäumen ebenfalls voneinander vollständig unabhängig. Die Wahrscheinlichkeit, an der Wurzel eine richtige heuristische Bewertung zu bekommen, ist ein Polynom in der Nichtfehlerwahrscheinlichkeit p.

Sei v exemplarisch ein MAX-Knoten. Die folgenden Skizzen verdeutlichen, wie man jene Polynome rekursiv berechnen kann.

1. v besitze den echten Wert 1:



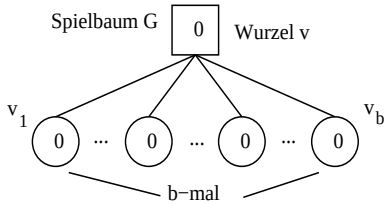
   Seien $v_1, \ldots, v_b$ die Nachfolger der Wurzel v der Beispielsituation in Abbildung 1. Seien $g_1(p), \ldots, g_b(p)$ vollständig unabhängige Wahrscheinlichkeiten dafür, daß die heuristischen Werte $h(v_i), i \in \{1 \ldots b\}$ mit den echten Werten $f(v_1) \ldots f(v_b)$ an den Knoten $v_1 \ldots v_b$ korrespondieren. Wir können dann die Wahrscheinlichkeit

$Q_v(p)$, daß der heuristische Minimaxwert von v mit dem echten Wert von v korrespondiert, folgendermaßen berechnen:
$Q_v(p)$ ist gleich der Wahrscheinlichkeit, daß nicht alle Nachfolger von v den heuristischen Wert 0 zugewiesen bekommen.

$$Q_v(p) = 1 - (\prod_{i=1}^{m}(1 - g_i(p)) \cdot \prod_{i=m+1}^{b} g_i(p))$$

Im folgenden sei *IN* die Menge der natürlichen Zahlen (ohne 0) und IR bezeichne die Menge der reellen Zahlen.

2. v besitzt den echten Wert 0:



Seien $v_1, \ldots, v_b$ wieder die Nachfolger des Knotens v, wobei v die Wurzel der Beispielsituation der Abbildung 2 sein soll. Wenn wir die Wahrscheinlichkeiten $g_1(p), \ldots, g_b(p)$ kennen, daß ein heuristischer Wert $h(v_i), i \in \{1 \ldots b\}$ mit dem echten Wert $f(v_i) = 0$ des Knotens v_i korrespondiert, können wir die Wahrscheinlichkeit $Q_v(p)$ dafür ausrechnen, daß der heuristische Wert h gleich dem echten Wert von v ist. Sie ist $Q_v(p) = \prod_{i=1}^{b} g_i(p)$.

Das führt uns zu folgender formalen Begriffsbildung:

Definition 4.3-14 **(Qualitätspolynom)**
Es sei $p \in [0, 1]$ die Nichtfehlerwahrscheinlichkeit der heuristischen Bewertungsfunktion am Blatt, es sei $G = ((V, E), f)$ Spielbaum mit Wurzel ϵ, und es sei $v \in V$. Für einen Knoten v gebe $m(v)$ an, wieviele Nachfolger von v den gleichen echten Wert wie v haben. Das *Qualitätspolynom* $Q_v(p)$ ist definiert durch

$$
\begin{aligned}
Q_v(p) &= p, && \text{falls } v \text{ Blatt ist} \\
Q_v(p) &= \prod_{i=1}^{b} Q_{v_i}(p), && \text{falls } v \text{ innerer MAX-Knoten mit Wert 0} \\
& && \text{oder innerer MIN-Knoten mit Wert 1 ist.} \\
Q_v(p) &= 1 - (\prod_{i=1}^{m(v)} (1 - Q_{v_i}(p)) \cdot \prod_{i=m(v)+1}^{b} Q_{v_i}(p)), \\
& && \text{falls } v \text{ innerer MAX-Knoten mit Wert 1} \\
& && \text{oder innerer MIN-Knoten mit Wert 0 ist.}
\end{aligned}
$$

$Q_\epsilon(p)$ nennen wir das Qualitätspolynom von G. □

Wir können formal einige interessante Sätze mit Hilfe einer neuen, sehr eleganten Technik beweisen (Beweise siehe nächstes Unterkapitel):

Sei $Q_\epsilon(p)$ das Qualitätspolynom eines Spielbaums G mit Wurzel ϵ. Bezeichne ferner $Q'_\epsilon(p)$ bzw. $Q^{(1)}_\epsilon(p)$ die erste Ableitung von $Q_\epsilon(p)$.

Lemma 4.3-1
Für alle Spielbäume gilt $Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$. □

Beobachtung 4.3-7
Schaut man in den Beweis von Satz 4.3-1, so sieht man auch die folgenden Eigenschaften des Qualitätspolynoms sofort ein: $Q'_\epsilon(1)$ ist immer ganzzahlig und nichtnegativ. $Q'_\epsilon(1)$ zählt, wieviele kritische Blätter der Spielbaum hat. (Dabei heißt ein Blatt kritisch, wenn eine Änderung seines echten Wertes bis zur Wurzel durchschlagen würde. [Alt00]) □

Satz 4.3-8
$Q'_\epsilon(1) = 0$ genau dann, wenn der Spielbaum G wenigstens zwei blattdisjunkte Beweisstrategien enthält, die den Wurzelwert von G beweisen. □

Sei G wieder ein beliebiger, endlicher Spielbaum und ϵ seine Wurzel. Mit $Q^{(n)}_\epsilon$ bezeichnen wir die n-te Ableitung von Q_ϵ. Durch Induktion können wir folgenden neuen, zentralen Satz beweisen:

Satz 4.3-9
$Q_\epsilon^{(n)}(1) = Q_\epsilon^{(n-1)}(1) = \cdots = Q_\epsilon^{(1)}(1) = 0 \Leftrightarrow$ Es gibt $n+1$ blattdisjunkte Beweisstrategien unter ϵ, die den echten Wert von ϵ beweisen. □

4.3.2 Interpretation der Ergebnisse

Abbildung 4.2 zeigt drei mögliche Kurvenverläufe von $Q_\epsilon(p)$ und zusätzlich die Identitätsfunktion. Wegen des folgendes Lemmas und wegen Lemma 4.3-1 sind diese in der Nähe von $p = 1$ auch alle Verläufe von Interesse.

Lemma 4.3-2
Sei Q_ϵ ein Qualitätspolynom des Spielbaums G. Dann gibt es ein $\delta > 0$, so daß $Q_\epsilon(p)$ im Intervall $p \in [1-\delta, 1]$ streng monoton steigt.

Grund: Q_ϵ ist ein Polynom. Sei n der endliche Grad dieses Polynoms. Damit hat Q'_ϵ höchstens $n-1$ verschiedene Nullstellen und Q_ϵ somit höchstens $n-1$ verschiedene lokale Extrema. Wir wissen über Q_ϵ, daß es für alle $\delta \in (0,1)$ ein $p \in [1-\delta, 1]$ gibt mit $Q_\epsilon(p) < 1$. (Sonst wäre $Q_\epsilon(p) \equiv 1$, was aufgrund der Konstruktion von Qualitätspolynomen ausgeschlossen werden kann, oder es gäbe ein $p \in [0,1]$ mit $Q_\epsilon(p) > 1$, was ebenfalls ausgeschlossen werden kann, weil Q_ϵ eine Wahrscheinlichkeit ist.) Wir sortieren nun die lokalen Extrema von Q_ϵ und nehmen uns das der 1 am nächsten liegende lokale Minimum heraus. Liege dieses an der Stelle δ mit $\delta < 1$. Für alle $p \in (1-\delta, 1)$ gilt nun, daß $Q'_\epsilon(p) \neq 0$ ist, und damit in diesem Fall, daß $Q'_\epsilon(p) > 0$ ist. Damit ist $Q_\epsilon(p)$ streng monoton steigend auf dem Intervall $(1-\delta, 1)$. □

Für den Fall, daß $Q_\epsilon(p) \leq p$, bietet eine direkte Bewertung offenbar eine bessere Chance, den wahren Wert der Wurzel von G zu erkennen, als die Minimax-Auswertung von G.
Das heißt, es ist nur dann sinnvoll einen Suchbaum auszuwerten, wenn gilt, daß $Q_\epsilon(p) > p$. Sonst wäre es besser, die Wurzel ϵ mit Hilfe der heuristischen Bewertungsfunktion direkt zu bewerten.

Beobachtung 4.3-8
Wenn $Q_{\epsilon_1}(p)$ und $Q_{\epsilon_2}(p)$ Qualitätspolynome von zwei Spielbäumen G und H sind und wenn gilt, daß $Q'_{\epsilon_1}(1) = 0$ und $Q'_{\epsilon_2}(1) \geq 1$ ist, gibt es immer ein $\delta > 0$, so daß für alle $p \in [1-\delta, 1)$ gilt, daß $Q_{\epsilon_1}(p) > Q_{\epsilon_2}(p)$ ist.
Wenn unsere Bewertungsfunktion gut genug ist, wird G zu besseren Erkennungsergebnissen führen als H. Man beachte, daß dies für alle Paare von Spielbäumen mit den ge-

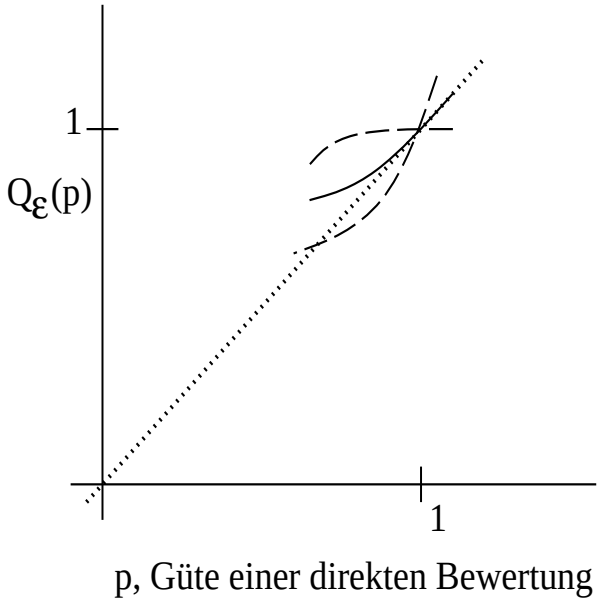


Abb. 4.2: Skizze möglicher Kurvenverläufe von $Q_\epsilon(p)$.

nannten Eigenschaften gilt! Dabei könnte G eine Hülle sein, die dem Schach-Mittelspiel entnommen ist, und H könnte gleichzeitig vom Damespiel entnommen worden sein. □

Nach diesen Überlegungen veranlassen uns Lemma 4.3-1 und Satz 4.3-8 zur folgenden Definition:

Definition 4.3-15 **(Nützlichkeit)**
$Q'_\epsilon(p)$ bzw. $Q^{(1)}_\epsilon(p)$ bezeichnen die erste Ableitung von $Q_\epsilon(p)$. Wir nennen einen Spielbaum G *nützlich*, wenn $Q'_\epsilon(1) = 0$. (Weil außerdem für alle Qualitätspolynome gilt, daß $Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$ ist (Satz 4.3-1), liefert uns diese Definition ein deutliches Unterscheidungskriterium zwischen 'guten' und 'schlechten' Spielbäumen, wenn nur unsere Bewertungfunktion gut genug ist.) □

Folgerung: Ein Spielbaum G ist somit genau dann nützlich, wenn G mindestens zwei blattdisjunkte Strategien enthält, die den echten Wert der Wurzel von G belegen.

Wenn wir nun einmal eine Taylorreihenentwicklung unserer Qualitätsploynome betrachten, stellen wir fest, daß die Reihenentwicklung $f(p) = f(1) + f'(1)(p-1) + \cdots + \frac{f^{(n)}(1)}{n!}(p-1)^n + R_{n+1}(p)$ dazu führt, daß $|Q_\epsilon(p) - Q_\epsilon(1)| = O((1-p)^{n+1})$ genau dann gilt, wenn die ersten n Ableitungen von $Q_\epsilon(p)$ an der Stelle 1 gleich 0 sind. Mit Satz 4.3-9

wissen wir, daß das genau dann der Fall ist, wenn G $n+1$ blattdisjunkte Strategien enthält.

Folgerung: In der Nähe der Stelle $p = 1$ bestimmt die Anzahl der vorhandenen Beweisstrategien größenordnungsmäßig die Qualität der Minimax-Auswertung.

Wenn also ein Spielbaum G mehr blattdisjunkte Beweisstrategien enthält als ein anderer Spielbaum H und wenn p nahe genug bei 1 liegt, ist die Wahrscheinlichkeit dafür, daß der heuristische Minimaxwert von H mit dem Wurzelwert übereinstimmt, kleiner als die entsprechende Wahrscheinlichkeit bei G. Dies ist unabhängig von der Anzahl der Knoten, die G und H enthalten, unabhängig davon, von welchen Spielen G und H entnommen wurden und unabhängig von weiteren Struktureigenschaften wie z.B. dem Verzweigungsgrad in G und H.

Wir kommen dadurch zu folgenden Vermutungen für die Praxis:

- Normalerweise werden wir nicht wissen, wieviele blattdisjunkte Beweisstrategien eine von uns ausgewählte Hülle enthält, da wir die echten Werte der Knoten nicht kennen. Man kann als Heuristik allerdings die blattdisjunkten Strategien zählen, die entstehen, wenn man die heuristischen Blattwerte auswertet. Man kann darauf hoffen, daß die durch die heuristischen Werte entstehende Struktur des Spielbaums die Struktur der echten Werte gut nachbildet. Genau das wird von der schon erwähnten Conspiracy Number Search gemacht. Sie maximiert die Anzahl der Blätter, die ihren heuristischen Wert ändern müssen, damit sich der heuristische Wert der Wurzel ändert. (Zum Zusammenhang von blattdisjunkten Strategien und Conspiracy Numbers siehe Satz 2.1-1.)

- Die Anzahl der den Wurzelwert beweisenden Strategien, die in einem Spielbaum G enthalten sind, ist nicht der Schlüssel zur Qualität von Spielbaumsuche, sondern die Anzahl der blattdisjunkten Beweisstrategien. Es kann durchaus Tausende von Beweisstrategien geben, aber zwei blattdisjunkte sind trotzdem nicht vorhanden.

- Der Verzweigungsgrad ist nicht Teil unserer Ergebnisse. Wir gehen deshalb davon aus, daß seine Bedeutung grob überschätzt wird, wenn es um die approximative Auswertung von Spielbäumen geht. Insbesondere für das Go-Spiel halten wir es zwar für denkbar, daß Spielbaumsuche in dem Spiel auch weiterhin nicht zu durchschlagendem Erfolg führen wird, man sollte aber den hohen Branchingfaktor des Spiels nicht als Grund anführen.

- Beim Schachspiel gibt es einige heuristik-basierte Ideen, wie man eine Hülle formen sollte. Wir glauben, daß der Effekt von Hüllen formenden Heuristiken, wie den

Fail High Reduktionen oder den Singular Extensions der ist, daß sie helfen, die Anzahl der blattdisjunkten Beweisstrategien bezüglich der echten Werte zu erhöhen.

Beispiele: Sei G der Spielbaum aus Abb. 4.1 (Seite 107). Sei H ein Spielbaum, der sich von G nur dadurch unterscheidet, daß der echte Wert des Knotens $v11$ eins ist. Dann gibt es keine zwei blattdisjunkten Strategien in H, die zeigen, daß der Wert der Wurzel von H null ist. Für $p = 0.9$ ergibt sich nun $Q_{\epsilon(H)}(p) = 0.85$ (auf 2 Stellen gerundet) und $Q_{\epsilon(G)}(p) = 0.97$. Abb.4.3 zeigt die jeweiligen Qualitätspolynome in Abhängigkeit von p.

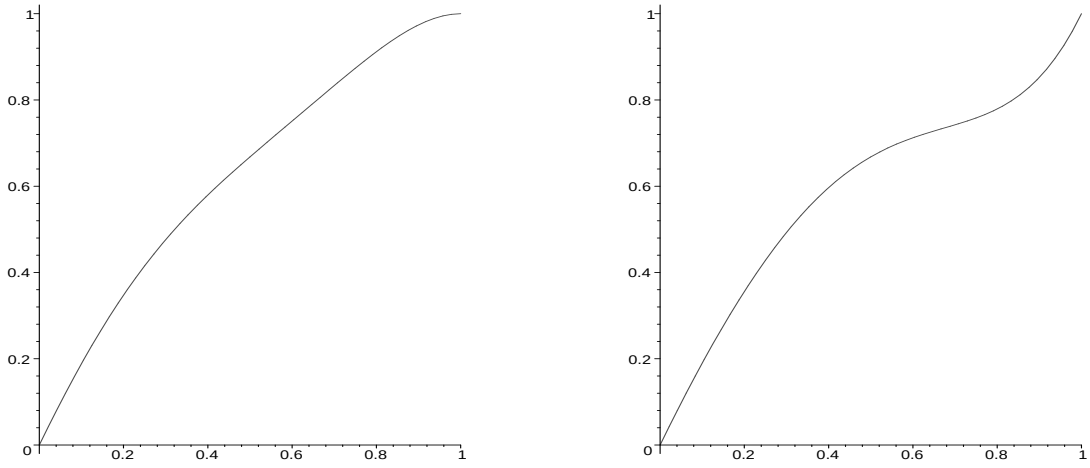


Abb. 4.3: $Q_{\epsilon(G)}(p)$ (links), $Q_{\epsilon(H)}(p)$ (rechts)

Für Tiefe-2-Spielbäume mit Verzweigungsgrad 100, die 10 blattdisjunkte (den Wurzelwert belegende) Strategien enthalten, oder für Tiefe-2-Spielbäume mit Verzweigungsgrad 30, die 3 blattdisjunkte Strategien enthalten, kann man sich eine Fehlerwahrscheinlichkeit von ungefähr 0,5% erlauben, ohne daß die Fehlerwahrscheinlichkeit einer Spielbaumauswertung schlechter wird als die einer direkten Bewertung.

Beweise für Lemma 4.3-1 und Satz 4.3-8

Definition 4.3-16 **(Tiefe eines Spielbaums)**
Die *Tiefe* eines Spielbaums G ist die maximale Distanz zwischen der Wurzel von G und seinen Blättern. □

Sei im folgenden $Q_\epsilon(p)$ das Qualitätspolynom eines Spielbaums G mit Wurzel ϵ. Bezeichne ferner $Q'_\epsilon(p)$ bzw. $Q^{(1)}_\epsilon(p)$ die erste Ableitung von $Q_\epsilon(p)$.

Lemma 4.3-1
Für alle Spielbäume gilt: $Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$.

Satz 4.3-8
$Q'_\epsilon(1) = 0$ gilt genau dann, wenn der Spielbaum G wenigstens zwei blattdisjunkte Beweisstrategien enthält, die beide den echten Minimaxwert der Wurzel von G beweisen.

Die entsprechenden Beweise werden über Induktion über die Tiefe von Spielbäumen geführt. Ohne Verlust von Allgemeingültigkeit befürchten zu müssen, betrachten wir Spielbäume, deren Wurzel ein MAX-Knoten ist. Für die anderen Spielbäume sind die Rechnungen analog. Bevor wir jedoch die Beweise führen, formulieren wir, um uns die Arbeit zu erleichtern, die ersten Ableitungen von Qualitätspolynomen in Abhängigkeit der Qualitätspolynome der Wurzelnachfolger. Wir unterscheiden dabei drei Fälle.

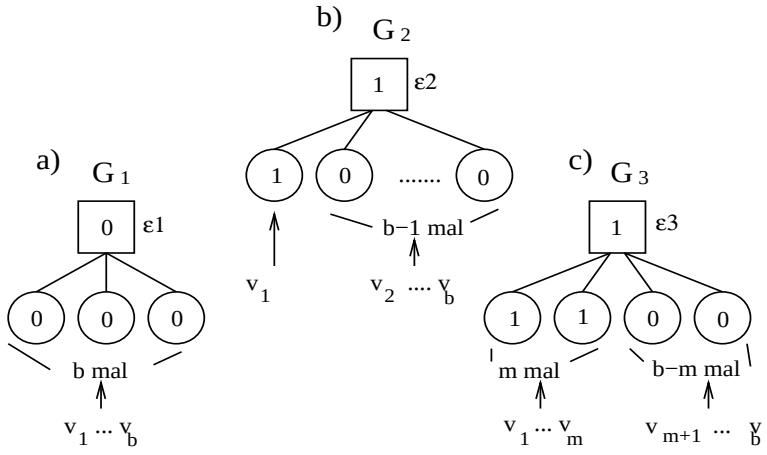


Abb. 4.4: Mögliche Konfigurationen an Spielbaumwurzeln

Abbildung 4.4a) zeigt einen Spielbaum mit einem MAX-Knoten als Wurzel und dem Wert 0. Wegen des Minimax-Prinzips haben auch alle Nachfolger den Wert 0. In Abbildung 4.4b) hat ein MAX-Knoten den Wert 1, und genau ein Nachfolger der Wurzel hat ebenfalls den Wert 1. In Abbildung 4.4c) gibt es mehr als einen Nachfolger, der wie der Wurzelknoten selber einen Wert 1 hat. Mit Hilfe der Überlegungen des vorangegangenen Abschnitts erhalten wir die Qualitätsploynome $Q_{\epsilon 1}$, $Q_{\epsilon 2}$ und $Q_{\epsilon 3}$ für die drei Typen von Situationen, wie sie durch G_1, G_2 und G_3 (Abbildung 4.4) aufgezeigt werden:

$$Q_{\epsilon 1}(p) = Q_{v_1}(p) \cdots Q_{v_b}(p)$$
$$Q_{\epsilon 2}(p) = 1 - (1 - Q_{v_1}(p)) \cdot Q_{v_2}(p) \cdots Q_{v_b}(p)$$
$$Q_{\epsilon 3}(p) = 1 - (1 - Q_{v_1}(p)) \cdots (1 - Q_{v_c}(p)) \cdot Q_{v_{c+1}} \cdots Q_{v_b}(p)$$

Die Ableitungen dieser drei Polynome lassen sich leicht angeben:
$Q'_{\epsilon 1}(p) = \sum_{i=1}^{b} Q_{v_1}(p) \cdots Q'_{v_i}(p) \cdots g_b(p)$
$Q'_{\epsilon 2}(p) = -(1-Q_{v_1}(p))' \cdot Q_{v_2}(p) \cdots g_b(p) + \sum_{i=2}^{b} (1-Q_{v_1}(p)) \cdot Q_{v_2}(p) \cdots Q'_{v_i}(p) \cdots Q_{v_b}(p)$
$Q'_{\epsilon 3}(p) = -\sum_{i=1}^{m} (1-Q_{v_1}(p)) \cdots (1-Q_{v_i}(p)) \cdots (1-Q_{v_c}(p)) \cdot Q_{v_{m+1}}(p) \cdots Q_{v_b}(p) -$
$\sum_{i=c+1}^{b} (1-Q_{v_1}(p)) \cdots (1-Q_{v_c}(p)) \cdot Q_{v_{c+1}}(p) \cdots Q'_{v_i}(p) \cdots Q_{v_b}(p)$

Weil $Q_{v_i}(1) = 1$ für alle i ist, folgt:
$Q'_{\epsilon 1}(1) = \underline{Q'_{v_1}(1) + \cdots + Q'_{v_b}(1)}$
$Q'_{\epsilon 2}(1) = \underline{Q'_{v_1}(1)}$
$Q'_{\epsilon 3}(1) = \underline{0}$

Der eigentliche Induktionsbeweis ist nun einfach (hier für Lemma 4.3-1 und Satz 4.3-8 gleichzeitig):

(IA) $t = 0 \Rightarrow Q'_\epsilon(p) = p \Rightarrow Q'_\epsilon(1) = 1$. Natürlich gibt es keine zwei blattdisjunkten Strategien in G.

(IV) Es gelte ($Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$), und es gelte ($Q'_\epsilon(1) = 0 \Leftrightarrow G$ enthält mindestens zwei blattdisjunkte Strategien, die beide den Wert der Wurzel belegen) für alle Spielbäume der Tiefe $t' < t$ mit Wurzel ϵ.

(IS) Es sei $G = ((V, E), f)$ ein Spielbaum mit Wurzel ϵ und Tiefe t. Es seien $v_1, \ldots, v_b$ die Nachfolger von ϵ in G. Dann unterscheiden wir drei Fälle, (wobei oBdA. ϵ ein MAX-Knoten ist):

1. f(ϵ) = 0. Dann sind die $f(v_i) = 0$ für alle $i \in \{1, \ldots, b\}$ und mit Hilfe unserer Vorüberlegungen gilt $Q'_\epsilon(1) = Q'_{v_1}(1) + \cdots + Q'_{v_b}(1)$. Mit der Induktionsvoraussetzung folgt sofort, daß $Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$ ist. Damit ist $Q'_\epsilon(1)$ genau dann 0, wenn alle Summanden $Q'_{v_i}(1)$ gleich 0 sind. Nach Induktionsvoraussetzung ist das genau dann der Fall, wenn es unter jedem Nachfolger von ϵ zwei blattdisjunkte Strategien gibt, die den Wert des jeweiligen Nachfolgers belegen. Das wiederum ist genau dann der Fall, wenn es unter ϵ mindestens zwei blattdisjunkte Strategien gibt, die den Wert 0 belegen.

2. Es sei $f(\epsilon) = 1$, und es gebe genau einen Nachfolger v_i von ϵ, für den $f(v_i) = 1$ ist. Dann ist $Q'_\epsilon(1) = Q'_{v_i}(1)$. Auch hier ist sofort klar, daß $Q'_\epsilon(1) = 0$ oder $Q'_\epsilon(1) \geq 1$ ist. Zwei oder mehr blattdisjunkte Strategien, die belegen, daß die Spielbaumwurzel den Wert 1 hat gibt es genau dann, wenn es zwei oder mehr blattdisjunkte Strategien

unter v_i gibt, die belegen, daß der Wert von v_i gleich 1 ist. Also gilt auch in diesem Fall die Aussage von Satz 4.3-8.

3. Falls $f(\epsilon) = 1$ ist und es mehrere Nachfolger der Wurzel von G gibt, die ebenfalls den Wert 1 haben, gilt für das Qualitätspolynom von G: $Q'_\epsilon(1) = 0$. Es gibt in diesem Fall auch mindestens zwei blattdisjunkte Strategien in G, die den Wert $f(\epsilon) = 1$ der Wurzel von G belegen.

Beweis von Satz 4.3-9

Sei G wiederum ein beliebiger Spielbaum, und sei ϵ seine Wurzel. $Q_\epsilon^{(n)}$ bezeichne von nun an die n-te Ableitung von Q_ϵ.

Vorüberlegungen

Lemma 4.3-3
Es sei $b \in I\!N$, es seien $f_1(x), \ldots, f_b(x)$ n-mal stetig differenzierbare Funktionen $\mathbb{R} \to \mathbb{R}$. Dann läßt sich die n-te Ableitung des Produkts der Funktionen darstellen als

$$(\prod_{i=1}^{b} f_i(x))^{(n)} = \sum_{y_1+\cdots+y_b=n} a(y_1, \ldots, y_b) \cdot f_1^{(y_1)}(x) \cdots f_b^{(y_b)}(x)$$

$$\text{mit passenden } a(y_1, \ldots, y_n) \in I\!N.$$

Die Koeffizienten der Form $a(0, \ldots, 0, n, 0, \ldots, 0)$ sind alle gleich 1. (ohne Beweis) □

Zu Satz 4.3-9:
$Q_G^{(n)}(1) = Q_G^{(n-1)}(1) = \cdots = Q_G^{(1)}(1) = 0 \Leftrightarrow$ Es gibt $n+1$ blattdisjunkte Beweisstrategien unter v, die alle den echten Wert von v beweisen.

Im Endeffekt möchten wir auch diesen Satz mit Hilfe von Induktion über die Höhe der Ableitungen n und der Tiefe t des Spielbaums G beweisen. Bevor wir uns jedoch daran wagen, untersuchen wir analog zum vorigen Abschnitt die n-ten Ableitungen von $Q_{\epsilon 1}, Q_{\epsilon 2}$ und $Q_{\epsilon 3}$ (vgl. Abbildung 4.4).

Wir berechnen $Q_{\epsilon 1}^{(n)}, Q_{\epsilon 2}^{(n)}$ und $Q_{\epsilon 3}^{(n)}$ unter folgenden drei Voraussetzungen:

(V_a) Für jeden Spielbaum G mit Wurzel ϵ und für alle $i \leq n$ gilt: $Q_\epsilon^{(i-1)}(1) = \cdots = Q_\epsilon^{(1)}(1) = 0 \Leftrightarrow$ Es gibt i blattdisjunkte Strategien unter ϵ, die den echten Wert $f(\epsilon)$ belegen.

(V_b) Für alle $j \in \{1, 2, 3\}$ gilt $Q_{\epsilon j}^{(n-1)}(1) = \cdots = Q_{\epsilon j}^{(1)}(1) = 0$, und es gibt n blattdisjunkte Strategien unter ϵj, die $f(\epsilon j)$ belegen.

(V_c) Für jeden Spielbaum G mit Wurzel ϵ und für alle $i \in \{1 \ldots n-1\}$ gilt

$$Q_\epsilon^{(i)}(1) \neq 0 \Rightarrow \text{ sgn}(Q_\epsilon^{(i)}(1)) = (-1)^{i-1}.$$

Bemerkung: Wenn wir später den Induktionsschluß durchführen, werden (V_a) und (V_c) die Induktionsvoraussetzung bilden. Den Induktionsschritt werden wir durchführen, indem wir die beiden Folgerungsrichtungen von Theorem 4.3-9 getrennt beweisen. (V_b) wird sich aus der jeweiligen Voraussetzung herleiten lassen: Entweder werden wir voraussetzen können, daß für jeden Spielbaum G $Q_\epsilon^{(n)}(1) = \cdots = Q_\epsilon^{(1)}(1) = 0$ gilt, oder daß es $n+1$ blattdisjunkte Beweisstrategien unter ϵ gibt, die den echten Wert $f(\epsilon)$ belegen. Die Ableitungen der Qualitätspolynome, die wir beim Beweis betrachten müssen, sind bei beiden Richtungen gleich.

Die n-ten Ableitungen von Qualitätspolynomen

Seien die drei Voraussetzungen V_a, V_b und V_c erfüllt.

1. Allgemein ist

   $$Q_{\epsilon 1}^{(n)}(p) = \sum_{y_1+\cdots+y_b=n} a(y_1, \ldots, y_b) \cdot Q_{v_1}^{(y_1)}(p) \cdots Q_{v_b}^{(y_b)}(p)$$

   mit passenden $a(y_1, \ldots, y_b) \in I\!N$ nach Lemma 4.3-3.

   Aufgrund von Voraussetzung (V_b) sind alle Summanden, die eine Ableitung enthalten, die kleiner als n ist, an der Stelle $p = 1$ gleich Null. Weil für alle i $Q_{v_i}(1) = 1$ ist, schließen wir, daß

   $\underline{Q_{\epsilon 1}^{(n)}(1) = \sum_1^b Q_{v_i}^{(n)}(1)}$

2. $$Q_{\epsilon 2}^{(n)}(p) =$$
   $$(-1) \cdot \sum_{y_1+\cdots+y_b=n} a(y_1, \ldots, y_n) \cdot (1 - Q_{v_1}(p))^{(y_1)} \cdot Q_{v_2}^{(y_2)}(p) \cdots Q_{v_b}^{(y_b)}(p)$$

mit passenden $a(y_1, \ldots, y_n) \in I\!N$.

Aufgrund von Voraussetzung V_b gilt, daß an der Stelle $p = 1$ nur ein Summand der Summe ungleich Null ist. Wir schließen daraus, daß gilt:

$\underline{Q_{\epsilon 2}^{(n)}(1) = Q_{v_1}^{(n)}(1)}$

3. Die n-te Ableitung von $Q_{\epsilon 3}^{(n)}(p)$ ist etwas komplizierter auszurechnen.

$$Q_{\epsilon 3}^{(n)}(p) = (-1) \cdot \sum_{y_1+\cdots+y_b=n} a(y_1, \ldots, y_b) \cdot \underbrace{(1 - Q_{v_1}(p))^{(y_1)} \cdots (1 - Q_{v_m}(p))^{(y_m)} \cdot Q_{v_{m+1}}^{(y_{m+1})}(p) \cdots Q_{v_b}^{(y_b)}(p)}_{=:S_{y_1,\ldots,y_b}(p)}$$

mit nach Lemma 4.3-3 passenden $a(y_1, \ldots, y_b) \in I\!N$.

Zur Berechnung von $Q_{\epsilon 3}^{(n)}(1)$ unterscheiden wir drei Fälle:

(a) Es sei $n < m$.
Dann gibt es in jedem Summanden $S_{y_1,\ldots,y_b}(p)$ mindestens einen Faktor der Form $(1 - Q_{v_1}(p))^{(0)}$. An der Stelle $p = 1$ ist also jeweils mindestens ein Faktor jedes Summanden gleich Null. Dann ist $\underline{Q_{\epsilon 3}^{(n)}(1) = 0.}$

(b) Es sei $n = m$.
Sei $S_{y_1,\ldots,y_b}(p)$ ein beliebiger Summand von $Q_{\epsilon 3}^{(n)}(p)$ an der Stelle $p = 1$.

Ist $y_l = 0$ für ein $l \in \{1, \ldots, m\}$, so ist $S_{y_1,\ldots,y_b}(1) = 0, weil 1 - Q_{v_l}(1) = 0$ ist.
Ist $y_l > 1$ für ein $l \in \{1 \ldots m\}$, so gibt es ein $l' \in \{1, \ldots, m\}$ mit $y_{l'} = 0$. Damit gilt ebenfalls $S_{y_1,\ldots,y_b}(1) = 0$.
Ist $y_l > 0$ für ein $l \in \{m+1, \ldots, b\}$, so existiert ebenfalls ein $l' \in \{1, \ldots, m\}$ mit $y_{l'} = 0$. Also ist $S_{y_1,\ldots,y_b}(1) = 0$.
Für alle anderen Summanden gilt: $S_{y_1,\ldots,y_b}(1) = (-1)^m \cdot \prod_{i=1}^{m} Q_{v_i}^{(1)}(1)$

Mit $n = m$ bekommen wir also heraus, daß es ein $k \in I\!N$ gibt, so daß $\underline{Q_{\epsilon 3}^{(n)}(1) = (-1) \cdot (-1)^n \cdot k \cdot \prod_{i=1}^{n} Q_{v_i}^{(1)}(1).}$ Das Vorzeichen von $Q_{\epsilon 3}^{(n)}(1)$ ergibt sich unter zu Hilfenahme von Voraussetzung V_c als $(-1)^{n+1}$.

(c) Sei $n > m$.
Sei $S_{y_1,\ldots,y_b}(p)$ wieder ein beliebiger Summand von $Q_{\epsilon 3}^{(n)}(p)$.

i. Falls es ein l gibt mit $l \leq m$ und $y_l = 0$, wissen wir, daß $S_{y_1,\ldots,y_b}(1) = 0$ ist (vgl. 3b).

ii. Falls es ein l gibt mit $l > m$ und $y_l > 0$, erhalten wir $\sum_{i=1}^{m} y_i \leq n-1$. $S_{y_1,\ldots,y_b}(p)$ hat in diesem Fall die Form $(1-Q_{v_1}(p))^{(y_1)} \cdots (1-Q_{v_m}(p))^{(y_m)} \cdot X,$ für ein $X \in \mathbb{R}$. Wegen Voraussetzung V_b gibt es n blattdisjunkte Strategien unter $\epsilon 3$, die $f(\epsilon 3)$ belegen. Aufgrund der Definition von Strategien wissen wir, daß die Summe aller Beweisstrategien unterhalb der Knoten $v_1 \ldots v_m$ ebenfalls n ist. Weil $\sum_{i=1}^{m} y_i \leq n-1$ ist, können wir schließen, daß es einen Nachfolger $v_i, i \in \{1 \ldots m\}$ gibt, unter dem mehr als y_i-viele blattdisjunkte Strategien hängen, die $f(v_i)$ belegen. Mit Hilfe der Voraussetzung V_a wissen wir somit, daß es zumindest ein $i \in \{1 \ldots m\}$ gibt, so daß $(1 - Q_{v_i}^{(y_i)}(p))$ an der Stelle $p = 1$ Null wird.

iii. Zu guter Letzt müssen wir noch den Fall abdecken, daß $\sum_{i=1}^{m} y_i \leq n$ ist und $\prod_{i=1}^{m} y_i > 0$ ist. In diesem Fall bekommen wir für passende $a(y_1 + \cdots + y_m, 0 \ldots, 0) \in I\!N$

$$Q_{\epsilon 3}^{(n)}(1) =$$

$$- \sum_{y_1+\cdots+y_m=n} a(y_1, \ldots, y_m, 0 \ldots, 0)(1-Q_{v_1})^{(y_1)}(1) \cdots (1-Q_{v_m})^{(y_m)}(1) =$$

$$\underline{(-1)^{m+1} \textstyle\sum_{y_1+\cdots+y_m=n} a(y_1, \ldots, y_m, 0 \ldots, 0) Q_{v_1}^{(y_1)}(1) \cdots Q_{v_m}^{(y_m)}(1).}$$

Sei nun $S_{y_1,\ldots,y_b}(1) \neq 0$. Dann gilt für alle $i \in \{1, \ldots, m\}$ $y_i \neq 0$ und $\sum_{i=1}^{m} y_i = n$. Es folgt

$$sgn(S_{y_1,\ldots,y_b}(1)) = (-1)(-1)^m \prod_{i=1}^{m} sgn(Q_{v_i}^{(y_i)}).$$

Nach Voraussetzung V_c ist $sgn(Q_{v_i}^{(y_i)}(1)) = (-1)^{y_i - 1}$.
Damit ist

$$\begin{aligned} sgn(S_{y_1}, \ldots, S_{y_b}) &= (-1)^{m+1} \cdot \prod_{i=1}^{m} (-1)^{y_i - 1} \\ &= (-1)^{m+1+\sum_{i=1}^{m}(y_i-1)} \\ &= (-1)^{1+\sum_{i=1}^{m} y_i} \\ &= (-1)^{n+1} \end{aligned}$$

Also ist auch $Q_{\epsilon 3}^{(n)}(1) = (-1)^{n+1}$.

Induktionsbeweis

Nach diesen Vorüberlegungen können wir Theorem 4.3.1-9 durch Induktion beweisen.
(IA) Den Induktionsanfang für $n = 1$ haben wir durch Satz 4.3-1 bereits gemacht.

(IV) Die Induktionsvoraussetzung sei die folgende:

Für alle $i \leq n$ gilt: $Q_\epsilon^{(i-1)}(1) = \cdots = Q_\epsilon^{(1)}(1) = 0 \Leftrightarrow$ Es gibt i blattdisjunkte $f(\epsilon)$ belegende Strategien unter der Wurzel ϵ von G. Außerdem gilt für die Vorzeichen der $Q_\epsilon^{(i)}(1)$: $Q_\epsilon^{(i)}(1) \neq 0 \Rightarrow Q_\epsilon^{(i)}(1) = (-1)^{i-1}$.

(IS) $(n-1 \to n)$
Wir beginnen den Induktionsschritt $(n \to n+1)$ mit '$\Leftarrow$'.
Es gebe $n+1$ blattdisjunkte Beweisstrategien unter ϵ, der Wurzel eines Spielbaums G. Es ist klar, daß es auch n blattdisjunkte Strategien gibt. Mit Hilfe der Induktionsvoraussetzung wissen wir, daß die Voraussetzungen (V_a), (V_b) und (V_c) erfüllt sind.

Für die Wurzel ϵ können die drei Fälle eintreten, die $\epsilon 1, \epsilon 2, \epsilon 3$ des vorigen Abschnitts entsprechen. Man sieht dort auch leicht, daß $Q_\epsilon^{(n)} \neq 0 \Rightarrow sgn(Q_\epsilon^{(n)}) = (-1)^{n+1}$.
Durch Induktion über die Höhe h von G erhält man:

1. $$Q_\epsilon^{(n)} = \sum_{i=1}^{b} \underbrace{Q_{v_i}^{(n)}}_{=0} = 0,$$
   denn für jeden Nachfolger v_i von ϵ gibt es $n+1$ blattdisjunkte Strategien, die $f(\epsilon) = 1$ belegen.

2. Es gebe genau einen Nachfolger der Wurzel ϵ, der wie ϵ selber den Wert 1 habe. ObdA sei dieses v_1.
   $$Q_\epsilon^{(n)} = \underbrace{Q_{v_1}^{(n)}}_{=0} = 0,$$

3. (a) $Q_\epsilon^{(n)}(1) = 0$.
   (b)
   $$Q_\epsilon^{(n)}(1) = (-1)^{(n+1)} \cdot k \cdot \prod_{i=1}^{m} Q_{v_i}^{(1)}(1) = 0,$$
   denn werden $n+1$ blattdisjunkte Strategien auf $m = n$ Nachfolger verteilt, so ex. ein Nachfolger v_i, dessen Wert $f(v_i)$ durch 2 blattdisjunkte Strategien belegt wird. Dann ist aber $Q_{v_i}^{(1)}(1) = 0$.

(c)

$$Q_\epsilon^{(n)}(1) = (-1)^{m+1} \cdot \sum_{y_1+\cdots+y_m=n} a(y_1,\ldots,y_m) Q_{v_1}^{(y_1)}(1) \cdot \cdots \cdot Q_{v_m}^{(y_m)}(1) = 0,$$

denn es ist $\sum_{i=1}^{m} y_i = n$. Werden nun $n+1$ blattdisjunkte Strategien auf m Nachfolger verteilt, so ex. ein Nachfolger v_i, dessen Wert $f(v_i)$ durch $n_i > y_i$ blattdisjunkte Strategien belegt wird. Dann ist aber $Q_{v_i}^{(y_i)}(1) = 0$.

Ganz ähnlich ist es für die andere Richtung '$\Rightarrow$':

Sei Q_ϵ das Qualitätspolynom eines Spielbaums G. Seien $Q_\epsilon^{(n)} = \cdots = Q_\epsilon^{(1)} = 0$. Offensichtlich gilt erst recht, daß $Q_\epsilon^{(n-1)} = \cdots = Q_\epsilon^{(1)} = 0$. Der Induktionsvoraussetzung können wir entnehmen, daß es n blattdisjunkte Beweisstrategien unter der Wurzel ϵ von G gibt, die alle den Wurzelwert von G beweisen. Deshalb sind die Voraussetzungen (V_a), (V_b), und (V_c) erfüllt.

Damit ist auch hier klar, daß $Q_\epsilon^{(n)} \neq 0 \Rightarrow sgn(Q_\epsilon^{(n)}) = (-1)^{n+1}$.
Für die Wurzel können die drei Fälle eintreten, die $\epsilon 1, \epsilon 2, \epsilon 3$ des vorigen Abschnitts entsprechen. Durch Induktion über die Höhe h von G erhält man:

1. $$0 = Q_\epsilon^{(n)}(1) = \sum_{i=1}^{b} Q_{v_i}^{(n)}(1) = 0$$

   Wegen $Q_{v_i}^{(n)}(1) = 0$ oder $sgn(Q_{v_i}^{(n)}(1)) = (-1)^{n-1}$ folgt $Q_{v_i}^{(n)}(1) = 0$ für alle $i \in \{1,\ldots,b\}$. Dann gibt es $n+1$ blattdisjunkte Strategien, die $f(v_i)$ belegen, $1 \leq i \leq b$. Damit gibt es $n+1$ blattdisjunkte Strategien, die $f(\epsilon)$ belegen.

2. Es gebe genau einen Nachfolger der Wurzel ϵ, der wie ϵ selber den Wert 1 habe. ObdA sei dieses v_1.
   $$0 = Q_\epsilon^{(n)}(1) = Q_{v_1}^{(n)}(1).$$
   Dann gibt es $n+1$ blattdisjunkte Strategien, die $f(v_1)$ belegen. Damit gibt es $n+1$ blattdisjunkte Strategien, die $f(\epsilon)$ belegen.

3. (a) $n < m$. Dann gibt es mindestens m blattdisjunkte Strategien, die $f(\epsilon)$ belegen. Damit gibt es $n+1$ blattdisjunkte Strategien, die $f(\epsilon)$ belegen.

   (b) $n = m$.
   $$0 = Q_\epsilon^{(n)}(1) = (-1)^{n+1} \cdot k \cdot \prod_{i=1}^{m} Q_{v_i}^{(1)}(1).$$

Dann ex. ein $i \in \{1, \ldots, m\}$ mit $Q_{v_i}^{y_i}(1) = 0$, somit ex. ein Nachfolger, unter dem sich zwei blattdisjunkte Strategien befinden, die $f(v_i)(= f(\epsilon))$ belegen. Die übrigen $m-1$ Nachfolger liefern uns $m-1$ weitere Strategien, die $f(\epsilon)$ belegen. Damit gibt es $m-1+2 = m+1 = n+1$ blattdisjunkte Strategien, die $f(\epsilon)$ belegen.

(c) $n > m$.

$$0 = Q_\epsilon^{(n)}(1) = (-1)^{m+1} \cdot \sum_{y_1+,\ldots,+y_m=n} \underbrace{a(y_1, \ldots, y_m) \prod_{i=1}^{m} Q_{v_i}^{y_i}(1)}_{S_{y_1,\ldots,y_m}(1)} .$$

Es ist $S_{y_1,\ldots,y_m}(1) = 0$ oder $sgn(S_{y_1,\ldots,y_m}(1)) = (-1)^{n+1}$, also $S_{y_1,\ldots,y_m}(1) = 0$ für alle $y_1, \ldots, y_m$. Dann ex. für jedes $S_{y_1,\ldots,y_m}(1)$ ein $i \in \{1, \ldots, m\}$ mit $Q_{v_i}^{(y_i)}(1) = 0$. Also ex. $y_i + 1$ blattdisjunkte Strategien unter v_i, die $f(v_i)$ belegen. Damit gibt es $1 + \sum_{i=1}^{m} y_i = n+1$ blattdisjunkte Strategien, die $f(\epsilon)$ belegen.

■

4.4 Zusammenfassung

Wir haben ein kombinatorisches Spielbaummodell präsentiert, welches Fehler einer heuristischen Bewertungsfunktion mit Hilfe von Münzwürfen modelliert. Die Nicht-Fehlerwahrscheinlichkeit eines heuristischen Minimaxwertes ist dann an der Wurzel ϵ eines Spielbaums G ein Polynom Q_ϵ. Wir konnten einen 1:1-Zusammenhang zwischen der Anzahl blattdisjunkter Beweisstrategien in G und den Ableitungen von $Q_\epsilon(1)$ beweisen. Wir konnten dadurch zeigen, daß die Anzahl der blattdisjunkten Beweisstrategien, die in einem Spielbaum G enthalten sind, größenordnungsmäßig die Güte des heuristischen Minimaxwertes bestimmt. Der Nutzen von Spielbaumsuche mit heuristischen Blattwerten wird einfach verstehbar. Das Modell ermutigt außerdem zu Conspiracy Number Suchen.

4.5 Nachtrag: Modell und Wirklichkeit

Eines der größten Geheimnisse unserer Welt ist der Zusammenhang von Modellen und der Wirklichkeit. Konsens gibt es lediglich darüber, daß ein von Menschen geschaffenes Modell generell nicht *die* Wirklichkeit beschreibt, sondern einen speziellen Aspekt

derselben. Somit kann ein nichtdeterministisches Modell durchaus Aspekte einer deterministischen Realität gut beschreiben. Z.B. ist ein Pseudo-Zufallszahlen-Generator ein deterministischer Algorithmus, aber er kann benutzt werden, um Effekte auszunutzen, die auch ein echter Zufallszahlen-Generator hervorrufen würde.

Bei probabilistischen Modellen, die versuchen, Unwissen durch die Annahme von Wahrscheinlichkeiten auszugleichen, ist es besonders schwer, ihren Wahrheitsgehalt zu prüfen. Nehmen wir uns einmal zwei Spieler $\mathcal{A}$ und $\mathcal{B}$ her. Spieler $\mathcal{A}$ führt Münzwürfe durch. Beide Spieler geben verdeckt einen Tip ab, ob Wappen oder Zahl kommt. Falls sie den gleichen Tip abgeben, ist die Runde ungültig. Geben beide Spieler erst ihren Tip ab, und wirft Spieler $\mathcal{A}$ dann die Münze, wird dieses Spiel von der Mehrzahl von Befragten sicherlich als fair bezeichnet werden. Nun wirft $\mathcal{A}$ die Münze erst, dann geben die Spieler ihren Tip ab. Im Moment des Tippens ist das Ergebnis nicht mehr zufällig. Oder doch? Ob es für Spieler $\mathcal{B}$ gut oder schlecht ist, die Münzwürfe auch weiterhin als zufällig anzunehmen, hängt davon ab, ob $\mathcal{A}$ Wissen über die gefallene Münze hat.

So ähnlich ist es auch in unserem Fall mit den heuristischen Bewertungen von Spielsituationen. Solange wir (und alle anderen) zwar eine Vorstellung davon haben, wieviele Fehler unsere Bewertungsfunktion enthält, aber niemand weiß, an welchen Stellen diese auftreten, ist eine probabilistische Modellierung nicht anfechtbar.

Eine Erweiterung des Modells der Art, daß man fragt, mit welcher Wahrscheinlichkeit der echte Wert eines Spielbaums 1 ist, wenn man eine 1 als heuristischen Wert errechnet hat, ist aber unseres Erachtens nach schon fragwürdig und macht wohl keinen Sinn mehr:

Wir sind bisher immer davon ausgegangen, daß ein konkreter Baum gegeben ist, dessen Wert es zu erkennen gilt. Die Wahrscheinlichkeit, daß der Wert 1 ist, ist entweder 1 oder 0. Die Würfel sind diesbezüglich schon gefallen. Um das Experiment wie angedeutet zu erweitern, müßte man von einem erweiterten stochastischen Modell ausgehen, in dem eine Verteilung über die Menge der möglichen Spielbäume gegeben ist. Wenn man dann davon ausgeht, daß eine Wahrscheinlichkeit dafür gegeben ist, daß der tatsächliche Wert eines Blattes 1 ist, wenn die heuristische Bewertung eine 1 erkennt, kann man nur dann weiterrechnen, wenn man die entstehenden Abhängigkeiten der Wahrscheinlichkeiten kennt. Außerdem widerstrebt es uns, den Wurzelwert eines konkreten Spielbaums, den wir untersuchen wollen, als noch nicht konkret vorhanden anzusehen.

Wir gehen davon aus, daß die von uns beschriebenen Struktureffekte (d.h., daß sich Bewertungsfehler dort häufen, wo man von einer konkreten Position startend in großen Suchbäumen nur wenige blattdisjunkte belegende Strategien findet) tatsächlich auftreten und präsentieren dafür Beispiele aus dem Bereich der Schachendspiele:

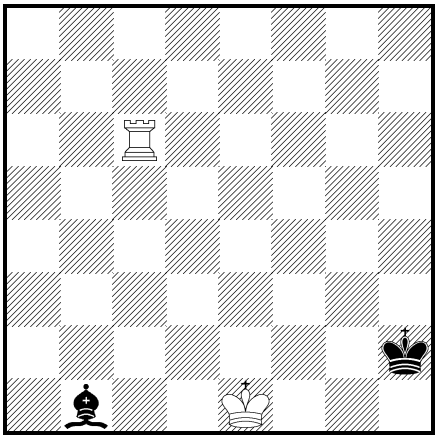


Abb. 4.5: Schachstellung

Im folgenden soll es unsere Aufgabe sein, den echten Wert einer Beispielstellung zu finden. (Es ist klar, daß man auch in der Lage ist, richtige Entscheidungen zu fällen, wenn man in der Lage ist, Werte von Stellungen richtig einzuschätzen.) Außerdem soll der heuristische Wert einer Stellung so berechnet werden, wie es im Computerschach üblich ist: Wir werten einen die Wurzel des Spielbaums enthaltenden Teilbaum mit Hilfe des Minimax-Prinzips aus. Die Werte der Blätter werden durch eine heuristische Bewertungsfunktion bestimmt, und die Werte von inneren Knoten sind die Minimaxwerte der Nachfolgestellungen.

Betrachten wir die Stellung von Abbildungen 4.5. Weiß ist dort am Zug, und wir würden gerne wissen, ob der echte Wert dieser Stellung remis ist oder ob Weiß gewinnen kann. Unsere heuristische Bewertung soll ein Endspiel der Form Turm gegen Läufer als remis einschätzen. Ein Endspiel Turm gegen nichts wird als Gewinnstellung für die Seite gewertet, die den Turm besitzt. Die Endspiele Läufer gegen nichts und König gegen König werden als remis bewertet. Das ist eine durchaus sinnvolle Art der Einschätzung und wird, sofern keine Endspieldatenbank zur Verfügung steht, auch von unserem Schachprogramm P.ConNerS als erste Näherung für einen heuristischen Wert benutzt.

Nun fügen wir Suche hinzu. Nach 1.Kf2 Kh3 2. Kf3 Kh4 3. Tc5 Lg6 [a) 3. ... La2 4. Kf4 Lb3 5. Tb5 Lc4 6. Tb4 Lf7 oder b) 3. ... Lh7 4. Kf4 Ld3 5. Td5 Lc4 6. Td4 Lb3 usw.] 4. Kf4 Kh3 5. Tg5 Lf7 6. Tg3+ Kh2 7. Tg3+ Kh2 8. Kf3 Lb3 9. Kf2 Lc2 10. Tc3 und einigen Verfeinerungen, die man mit Hilfe eines Schachprogramms schnell nachvollziehen kann, kommen wir zu dem Schluß, daß die Startstellung wohl doch für Weiß gewonnen ist. Die Kombination unserer Bewertungsfunktion plus der ausgewählte Suchbaum führt uns also zu der richtigen Einschätzung. Da die Beispielstellung aus einem Endspiel entnommen ist, für das es mittlerweile vollständige Datenbanken gibt, können wir bestätigen, daß der

Wert der Stellung in Abbildung 4.5 tatsächlich ein Gewinn für Weiß ist.

Um gemeinsam mit der vorausgesetzten heuristischen Bewertungsfunktion zu dem richtigen Ergebnis zu kommen, muß man einige sehr tiefe Varianten mitberücksichtigen. Betrachtet man die Verteilung der echten Werte, ausgehend von der gezeigten Startstellung, fällt auf, daß man die oben ausgeführten Varianten in den Suchbaum aufnehmen muß, wenn man zwei blattdisjunkte Strategien, die den Wert der Wurzel belegen, im Suchbaum haben möchte.

Betrachten wir zum Schluß noch das Endspiel König + Turm vs. König + Läufer (KTKL: am Zug ist der Spieler mit dem Turm) unter quantitativeren Gesichtspunkten: Für einige Endspiele (z.B. KTKL) haben wir eine Komplettdatenbank erzeugt. Unsere KTKL-Datenbank enthält 5390364 Stellungen. Man kann die Datenbank sicherlich besser packen, indem man weitere Spiegelungen beachtet; die absolute Größe der Datenbank ist aber für die folgenden Untersuchungen ohne Belang. Wir haben auf allen 5390364 Einzelstellungen Tiefe-t-Suchen ($t \in \{0, \ldots, 5\}$) gestartet. Die Suchen wurden danach unterteilt, ob sie zwei blattdisjunkte Strategien (bzgl. der echten Werte, die aus der Datenbank bekannt sind) enthalten, die den jeweiligen Stellungswert belegen, oder ob sie sie nicht enthalten.

Auch wenn hier ein Endspiel 'vollständig' ausgewertet wird, handelt es sich auch bei diesen Betrachtungen um ein Beispiel: Es gibt viele Bewertungsfunktionen, die man nutzen kann, und es gibt viele Arten, Suchbäume aufzuspannen. Allerdings wäre es doch sehr schade, wenn man sich die erstbeste Bewertungsfunktion hernehmen könnte, und auf der gegebenen Datenbank würden die Suchergebnisse besser ausfallen, wenn die Suchbäume keine zwei blattdisjunkte Strategien enthalten.

Bei einer Tiefe-0-Suche handelt es sich um eine direkte Bewertung (s.o.), und es gibt nur eine belegende Strategie. Die direkte heuristische Bewertung gibt 64,9% richtige Einschätzungen. Tabelle 4.6 zeigt den Prozentsatz der richtigen Einschätzungen der Tiefe-t-Suchen, jeweils unterteilt in Tiefe-t-Suchbäume, die mindestens zwei blattdisjunkte, den echten Wert belegende Strategien enthalten (2 bdS), und in Tiefe-t-Suchbäume, die keine solchen blattdisjunkten Strategien enthalten (keine 2 bdS).

Wie man sieht, sind die Suchbäume, die blattdisjunkte Strategien enthalten, immer erfolgreicher als diejenigen ohne blattdisjunkte Strategien. Natürlich ist es so, daß mit steigender Suchtiefe die Erfolgsquote der Tiefe-t-Suchen insgesamt steigt. Das wird in diesem Fall z.B. auch dadurch verursacht, daß der Zieher Strategien finden kann, die entweder in das Endspiel KTK oder KKL umlenken. Diese Endspiele werden nahezu perfekt von der hier vorgestellten heuristischen Bewertungsfunktion eingeschätzt. Allerdings steigt eben

Erfolgsquoten von Tiefe-t-Suchen im Endspiel KTKL

Suchtiefe	keine 2 bdS	2 bdS	Gesamt
0	64,9%	-	64,9%
1	72,4%	95,2%	90,0%
2	72,6%	98,4%	92,0%
3	90,0%	92,6%	92,0%
4	88,2%	98,7%	98,0%
5	67,7%	97,6%	97,4%

Abb. 4.6: Wirkung blattdisjunkter Strategien (bdS)

auch der Anteil derjenigen Tiefe-t-Suchbäume, die zwei oder mehr blattdisjunkte Strategien enthalten. Ähnliche Ergebnisse (wenn auch nicht immer ganz so deutlich) bekommt man auch, wenn man die Seite mit dem Läufer zuerst ziehen läßt oder wenn man z.B. die Endspiele König + Springer vs. König + Turm betrachtet.

Erfolgsquoten von Tiefe-t-Suchen im Endspiel KLKT

Suchtiefe	keine 2 bdS	2 bdS	Gesamt
0	96,7%	-	96,7%
1	41,4%	99,9%	97,0%
2	86,4%	98,2%	98,0%
3	85,6%	99,6%	99,0%
4	44,4%	99,0%	99,0%
5	36,2%	99,5%	99,4%

Erfolgsquoten von Tiefe-t-Suchen im Endspiel KTKS

Suchtiefe	keine 2 bdS	2 bdS	Gesamt
0	51,6%	-	51,6%
1	56,2%	82,0%	77,2%
2	56,0%	84,8%	79,3%
3	60,0%	82,0%	80,3%
4	59,4%	88,8%	86,5%
5	31,3%	99,5%	89,4%

Kapitel 5

Zusammenfassung und offene Probleme

Der zentrale Begriff dieser Arbeit ist der der *blattdisjunkten Strategien*. Die vorgestellten Algorithmen und Analysen geben aus verschiedenen Blickwinkeln Hinweise darauf, daß blattdisjunkte Strategien eine zentrale Bedeutung in der Spielbaumsuche haben.

1. Der vorgestellte 'Controlled Conspiracy Number Search Algorithmus' ist in der Lage, effizient Spielbäume aufzubauen und zu durchsuchen. Er sucht nach Spielbäumen und wertet diese aus, in denen die untere Schranke für den Wert des besten Wurzelnachfolgers sowie obere Schranken für die Werte der übrigen Wurzelnachfolger jeweils durch mehrere blattdisjunkte Strategien belegt werden. Die Blätter dieser Strategien müssen zusätzlich eine vorgegebene Mindestentfernung zur Wurzel haben.

2. Der 'Parallel Controlled Conspiracy Number Search Algorithmus', den wir vorgestellt haben, bettet seinen Spielbaum in ein Prozessornetzwerk ein. Auf einem SCI-Workstationcluster wurden mit 160 Prozessoren Effizienzen von über 30% erreicht. In Anbetracht der Tatsachen, daß es sich dabei um einen Workstationcluster und nicht um einen klassischen Parallelrechner handelt und daß nicht nur die Last, sondern auch der Speicher verteilt werden muß, ist das Ergebnis beachtlich.

3. In einem Spielbaummodell, in dem Fehler einer nicht perfekten Bewertungsprozedur durch Zufallsereignisse modelliert werden, konnten wir zeigen, daß die Fehlerwahrscheinlichkeit einer Minimaxauswertung eines Spielbaums größenordnungsmäßig von der Anzahl blattdisjunkter Strategien abhängt, die im Spielbaum bezüglich seiner echten Werte enthalten sind und den Wert der Wurzel belegen.

Diese Arbeit erhält besondere Bedeutung dadurch, daß sich die hier beschriebenen Algorithmen im Schachprogramm P.ConNerS bewährt haben. Der vielbeachtete Gewinn des 10. Lippstädter Großmeisterturniers zeigt, daß der Controlled Conspiracy Number Search

Algorithmus eine sehr gute Alternative zu den reinen Tiefensuchalgorithmen wie dem $\alpha\beta$-Algorithmus darstellt.

Offene Probleme:

a) Beim parallelen $\alpha\beta$-Algorithmus wurden auch für 256 Prozessoren noch Effizienzen von 49% erreicht. Lassen sich solche Effizienzen auch beim Controlled Conspiracy Number Search Algorithmus erreichen?

b) Bei unseren Fehleranalysen in Spielbäumen hängen Bewertungsfehler der Wurzel immer von der Struktur der Spielbäume bezüglich der echten Werte dieser Spielbäume ab. In der Praxis kennt man diese Werte nicht, und wir haben Conspiracy Number Search deshalb als Suchheuristik eingestuft. Kann man das Modell so erweitern, daß man den Nutzen von Conspiracy Number Search direkt nachweisen kann? Kann man vielleicht auch nachweisen, daß es von Vorteil sein kann, an ALL-Knoten nicht immer alle Nachfolger zu betrachten?

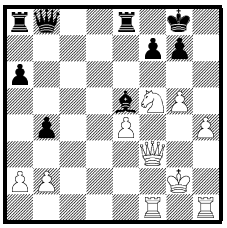
c) Die in dieser Arbeit vorgestellten Verfahren und Analysen beziehen sich alle auf Zweipersonen-Nullsummenspiele. Lassen sie sich auch auf Mehrpersonenspiele, (möglicherweise) ohne Nullsumme erweitern und übertragen?

Danksagung:

An dieser Stelle möchte ich mich bei meinem Betreuer Prof. Dr. Burkhard Monien bedanken, der durch seine langährige Unterstützung sowohl diese Arbeit als auch das Schachprogramm 'P.ConNerS' erst möglich gemacht hat.
In gleicher Weise gilt mein Dank meinem Kollegen Dr. Rainer Feldmann, der in unzähligen Diskussionen und Gesprächen mit hilfreichen Kommentaren zum Erfolg beigetragen hat.
Des weiteren bedanke ich mich bei Prof. Dr. Ingo Althöfer für seine wertvollen Anregungen und Korrekturen, sowie bei meinen Eltern für einige sprachliche Hinweise.

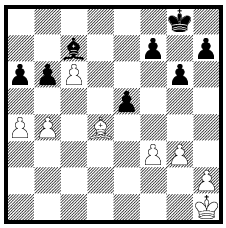
Der BT2630-Test

BT01



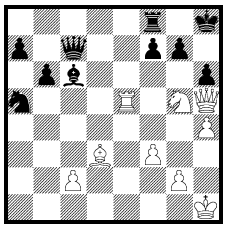
−− f5g7 −−

BT02



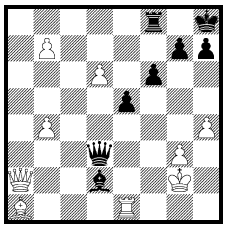
−− d4b6 −−

BT03



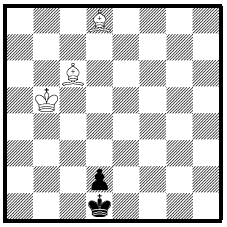
−− e5e6 −−

BT04



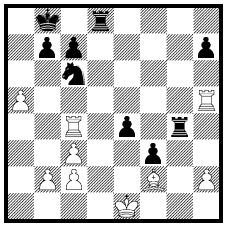
−− a2f7 −−

BT05



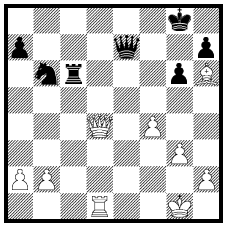
−− b5a6 −−

BT06



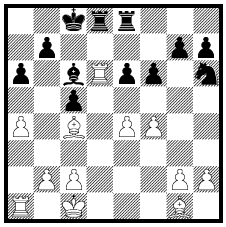
−− e4e3 −−

BT07



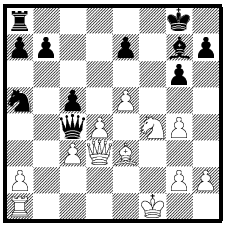
−− c6d6 −−

BT08



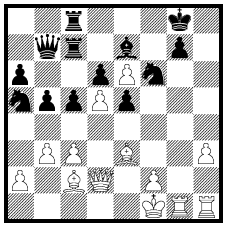
−− d6c6 −−

BT09



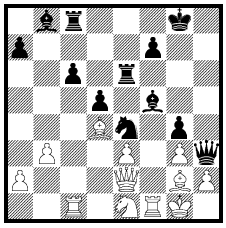
−− g6g5 −−

BT10



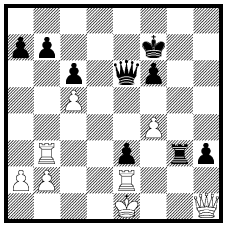
−− g1g7 −−

BT11



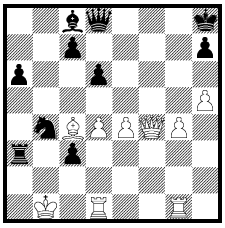
−− h3h2 −−

BT12



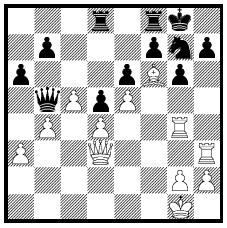
−− e6e4 −−

BT13a



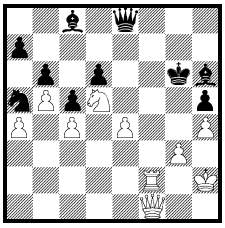
−− c8e6 −−

BT14



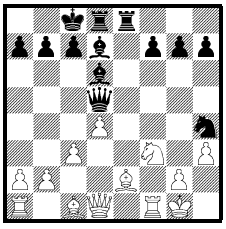
−− h3h7 −−

BT15a



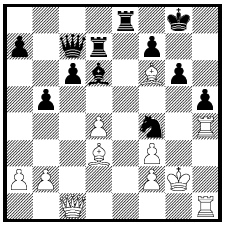
−− e4e5 −−

BT16a



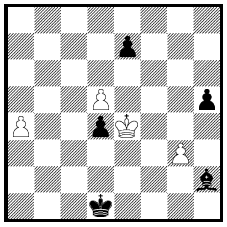
−− h4g2 −−

BT17



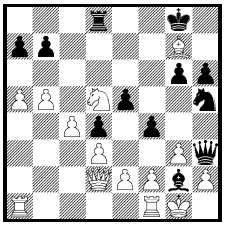
−− c1f4 −−

BT18



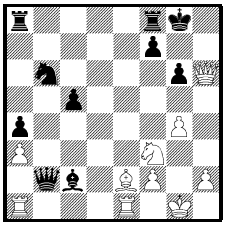
−− d5d6 −−

BT19



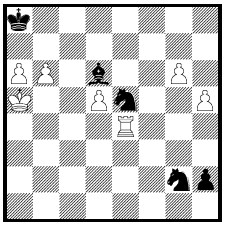
−− f4f3 −−

BT20



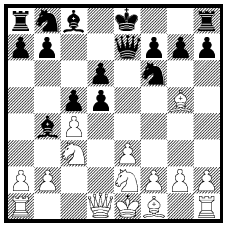
−− a1a2 −−

BT21a



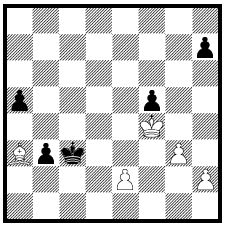
−− e4e1 −−

BT22



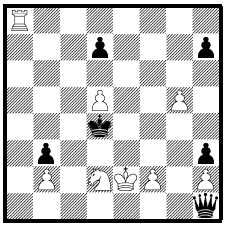
++ a2a3 ++

BT23a



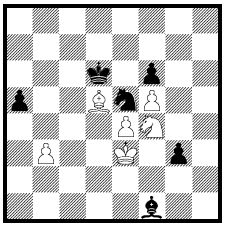
−− g3g4 −−

BT24



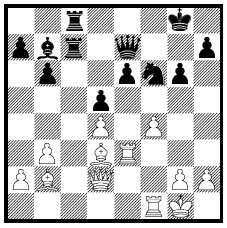
−− g5g6 −−

BT25



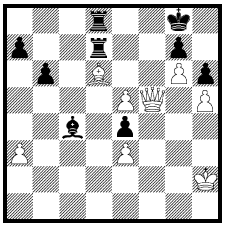
−− e5d3 −−

BT26



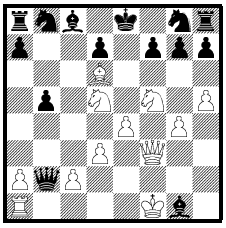
−− f4f5 −−

BT27



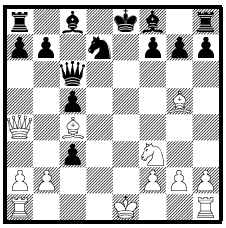
−− e5e6 −−

BT28a



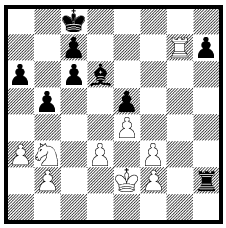
−− e4e5 ++

BT29a



++ e1c1 ++

BT30



−− f3f4 −−

Literaturverzeichnis

[AD87] *G. M. Adelson-Velsky, V. L. .Arlazarov, M. V. Donskoy.* **Algorithms for Games**. Springer, (1987).

[ADLR93] *I. Althöfer, C. Donninger, U. Lorenz, and V. Rottmann.* **On Timing, Permanent Brain and Human Intervention.** Advances in Computer Chess 7 (ed J. van den Herik), (1993).

[All94] *L. V. Allis.* **Searching for Solutions in Games and Artificial Intelligence**. (1994). Doctoral-Thesis, University of Limburg, Maastricht, The Netherlands.

[Alt88] *I. Althöfer.* **Root Evaluation Errors: How They Arise and Propagate.** ICCA Journal, 11(3):55–63, (1988).

[Alt90a] *I. Althöfer.* **Generalized Minmax Algorithms are no better Error Correctors than Minmax Itself.** In D.F. Beal, editor, Advances in Computer Chess 5, pages 265–282. North-Holland, (1990).

[Alt90b] *I. Althöfer.* **An Incremental Negamax Algorithm.** Artificial Intelligence, 43(1):57–65, (1990).

[Alt00] *I. Althöfer.* Email-Korrespondenz. (2000).

[Ana91] *T.S. Anantharaman.* **Extension Heuristics.** ICCA Journal, 14(2):47–63, (1991).

[Bau78] *G.M. Baudet.* **The Design and Analysis of Algorithms for Asynchronous Multiprocessors**. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, (1978).

[Bea80] *D.F. Beal.* **An Analysis of Minimax.** In M.R.B. Clarke, editor, Advances in Computer Chess 2, pages 103–109. Edinburgh University Press, (1980).

[Bea82] *D.F. Beal.* **Benefits of Minimax Search.** In M.R.B. Clarke, editor, Advances in Computer Chess 3, pages 17–24. Pergamon Press, (1982).

[Bea89] *D.F. Beal.* **Experiments with the Null Move.** Advances in Computer Chess 5 (ed. Beal, D.F.), pages 65–79, (1989).

[Bea99] *D. F. Beal.* **The Nature of Minimax Search**. (1999). Doctoral-Thesis, University of Maastricht, The Netherlands.

[Ber79] *H. Berliner.* **The B* Tree Search Algorithm: A Best-first Proof Procedure.** Artificial Intelligence, 12(1):23–40, (1979).

[BG82] *I. Bratko and M. Gams.* **Error Analysis of the Minimax Principle.** In M.R.B. Clarke, editor, Advances in Computer Chess 3, pages 1–15. Pergamon Press, (1982).

[BJK+95] *R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, A. Shaw, and Y. Zhou.* **Cilk: An Efficient Multithreaded Runtime System.** Proceeding of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, (1995).

[BT94] *H. Bednorz and F. Tönissen.* **Der neue Bednorz-Tönissen-Test.** Computer Schach und Spiele, 11(2):24–27, (1994).

[CT82] *J.H. Condon and K. Thompson.* **Belle Chess Hardware.** Advances in Computer Chess III, M.R.B. Clarke (Editor), Pergamon Press, pages 44–54, (1982).

[Don93] *C. Donninger.* **Null Move and Deep Search.** ICCA Journal, 16(3):137–143, (1993).

[Don95] *C. Donninger.* Persönliches Gespräch. (1995).

[Don00] *C. Donninger.* **Die Wissenschaft kehrt zurück.** Computer-Schach & Spiel, Heft 2, Seite 40, (2000).

[EBG82] *J. H. Conway E. Berlekamp and R. K. Guy.* **Winning Ways for Your Mathematical Plays.** Academic Press, London, England, (1982).

[Fel93] *R. Feldmann.* **Spielbaumsuche mit massiv parallelen Systemen**. (1993). Doctoral-Thesis, University of Paderborn, Germany.

[Fel96] *R. Feldmann.* **Fail High Reductions.** Advances in Computer Chess 8 (ed. J. van den Herik), (1996).

[FF82] *R.A. Finkel and J.P. Fishburn.* **Parallelism in Alpha-Beta Search.** Artificial Intelligence, 19(1):89–106, (1982).

[FK88] *C. Ferguson and R.E. Korf.* **Distributed Tree Search and Its Application to Alpha-Beta Pruning.** In Proceedings of the Seventh National Conference Artificial Intelligence (AAAI-88), pages 128–132, Minneapolis, MN, August (1988).

[FMM91] *R. Feldmann, P. Mysliwietz, and B. Monien.* **Distributed Game Tree Search on a Massively Parallel System.** In B. Monien and Th. Ottmann, editors, Data structures and efficient algorithms: Final report on the DFG special joint initiative, volume LNCS 594, pages 270–288. Springer-Verlag, September (1991).

[FMMV90] *R. Feldmann, B. Monien, P. Mysliwietz, and O. Vornberger.* **Distributed Game Tree Search.** In L.N. Kanal V. Kumar, P.S. Gopalakrishnan, editor, Parallel Algorithms for Machine Intelligence and Vision, pages 66–101. Springer-Verlag, (1990).

[FO88] *E.W. Felten and S.W. Otto.* **Chess on a Hypercube.** In G. Fox, editor, The Third Conference on Hypercube Concurrent Computers and Applications, volume II-Applications, pages 1329–1341, Passadena, CA, (1988).

[Fra95] *A. S. Fraenkel.* **Combinatorial Games: Selected Bibiliography with a Succinct Gourmet Introduction.** MSRI Publications, 29, (1995).

[Gas95] *R. U. Gasser.* **Harnessing Computational Recources for Efficient Exhaustive Search**. (1995). Doctoral-Thesis, Swiss Federal Institute of Technology Zürich, Swiss.

[GEIC67] *R.D. Greenblatt, D.E. Eastlake III, and S.D. Crocker.* **The Greenblatt Chess Program.** Proceeding of the Fall Joint Computing Conference, San Francisco, pages 801–810, (1967).

[Hei99] *E. A. Heinz.* **Scalable Search in Computer Chess**. (1999). Doctoral-Thesis, University of Karlsruhe, Germany.

[HGN85] *R.M. Hyatt, B.E. Gower, and H.L. Nelson.* **Cray Blitz.** Advances in Computer Chess IV, D.F. Beal (Editor), Pergamon Press, pages 8–18, (1985).

[HH99] *H. Hellwagner and A. Reinefeld* . **SCI:Scalable Coherent Interface: Architecture and Software for High-Performance Compute Clusters**. Lecture Notes in Computer Science 1734, (1999).

[HS89] *R. M. Hyatt and B. W. Suter.* **A Parallel Alpha/Beta Tree Searching Algorithm.** Parallel Computing, 10:299–308, (1989).

[Hsu90] *F-H. Hsu.* **Large Scale Parallelization of Alpha-Beta Search: An Algorithmic and Architectural Study with Computer Chess**. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, February (1990).

[JS97] *A. Plaat J. Schaeffer.* **Kasparov versus Deep Blue: The Rematch.** ICCA Journal, pages 95–101, (1997).

[KM75] *D.E. Knuth and R.W. Moore.* **An Analysis of Alpha-Beta Pruning.** Artificial Intelligence, 6(4):293–326, (1975).

[LN91] *D. Levy and M. Newborn.* **How Computers Play Chess**. (1991).

[LR96] *U. Lorenz and V. Rottmann .* **Parallel Controlled Conspiracy Number Search.** Advances in Computer Chess 8 (ed J. van den Herik), (1996).

[Lor00] *U. Lorenz.* **Controlled Conspiracy-2 Search.** Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science (STACS), (H. Reichel, S.Tison eds), Springer LNCS, pages 466–478, (2000).

[LRFM95] *U. Lorenz, V. Rottmann, R. Feldmann, and P. Mysliwietz.* **Controlled Conspiracy Number Search.** ICCA Journal, 18(3):135–147, (1995).

[LVAvdH89] *M. van der Meulen L. V. Allis and H. J. van den Herik.* **A Knowledge-Based Approach to Connect-Four: The Game is over: White To Move Wins.** Heuristik Programming in Artificial Intelligence I, The First Computer Olympiad, (1989).

[MC82] *T.A. Marsland and M.S. Campbell.* **Parallel Search of Strongly Ordered Game Trees.** ACM Computing Surveys, 14(4):533–551, December (1982).

[McA88] *D.A. McAllester.* **Conspiracy Numbers for Min-Max Searching.** Artificial Intelligence, 35(1):287–310, (1988).

[MOS86] *T.A. Marsland, M. Olafsson, and J. Schaeffer.* **Multiprocessor Tree-Search Experiments.** In D.F. Beal, editor, Advances in Computer Chess 4, pages 37–51. Pergamon Press, (1986).

[MRS87] *T.A. Marsland, A. Reinefeld, and J. Schaeffer.* **Low Overhead Alternatives to SSS*.** Artificial Intelligence, 31(1):185–199, (1987).

[Mys94] *P. Mysliwietz.* **Konstruktion und Optimierung von Bewertungsfunktionen beim Schach**. (1994). Doctoral-Thesis, University of Paderborn, Germany.

[Nau79] *D.S. Nau.* **Quality of Decision versus Depth of Search on Game Trees**. PhD thesis, Duke University, Durham, NC, (1979).

[Nau82] *D.S. Nau.* **An Investigation of the Causes of Pathology in Games.** Artificial Intelligence, 19(3):257–278, (1982).

[New88] *M. Newborn.* **Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search.** IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-10(5):687–694, (1988).

[Pal85] *A.J. Palay.* **Searching with Probabilities**. (1985).

[Pea83] *J. Pearl.* **On the Nature of Pathology in Game Searching.** Artificial Intelligence, 20(4):427–453, (1983).

[Pea84] *J. Pearl.* **Heuristics – Intelligent Search Strategies for Computer Problem Solving**. Addison-Wesley Publishing Co., Reading, MA, (1984).

[Pla93] *A. Plaat.* **RESEARCH, RE:SEARCH & RE-SEARCH**. (1993). Doctoral-Thesis, University of Rotterdam, Netherlands.

[Rei89] *A. Reinefeld.* **Spielbaum - Suchverfahren**. Springer, (1989).

[Riv87] *R.L. Rivest.* **Game Tree Searching by Min/Max Approximation.** Artificial Intelligence, 34(1):77–96, (1987).

[Sch83] *J. Schaeffer.* **The History Heuristic.** ICCA Journal, 6(3):16–19, (1983).

[Sch86] *G. Schrüfer.* **Presence and Absence of Pathology on Game Trees.** In D. F. Beal, editor, Advances in Computer Chess 4, pages 101–112. Pergamon Press, (1986).

[Sch88] *G. Schrüfer.* **Minimax-Suchen Kosten, Qualität und Algorithmen**. Doctoral-Thesis, University of Braunschweig, Germany, (1988).

[Sch89a] *J. Schaeffer.* **Distributed Game-Tree Searching.** Journal of Parallel and Distributed Computing, 6:90–114, (1989).

[Sch89b] *J. Schaeffer.* **The History Heuristic and Alpha-Beta Search Enhancements in Practice.** IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-11(1):1203–1212, November (1989).

[Sch90] *J. Schaeffer.* **Conspiracy Numbers.** *Artificial Intelligence*, 43(1):67–84, (1990).

[Sei97] *Y. Seirawan.* **The Kasparov - Deep Blue Games.** ICCA Journal, pages 102–125, (1997).

[Sel91] *Reinhard Selten.* **Game Equilibrium Models III**. Springer-Verlag Berlin Heidelberg, (1991).

[Sha50] *C.E. Shannon.* **Programming a Computer for Playing Chess.** Philosophical Magazine, 41:256–275, (1950).

[Sto79] *G.C. Stockman.* **A Minimax Algorithm Better than Alpha-Beta?** Artificial Intelligence, 12(2):179–196, (1979).

[Tur53] *A.M. Turing.* **Digital Computers Applied to Games.** B.V. Bowden: Faster than Thought: A Symposium on Digital Computing Machines, Pitman, pages 286–310, (1953).

[vdM90] *M. van der Meulen.* **Conspiracy Number Search.** ICCA Journal, 13(1):3–14, March (1990).

[vNM43] *J. von Neumann and O. Morgenstern.* **Theory of Games and Economic Behaviour**. Princeton University Press, (1943).

[WE96] *J.-C. Weill.* **The ABDADA Distributed Minimax-Search Algorithm**. , ICCA Journal, 19(1):3–14, (1996).

[Zus84] *K. Zuse.* **Der Computer — Mein Lebenswerk.** Springer Verlag, (1984).

Zeitungsartikel

[BILD00] *K. Wessels.* **Das Schachmonster.** BILD-Zeitung, 20. Juli (2000).

[CB00] *K. Müller.* **P.Conners gewinnt als erster Rechner ein GM-Turnier.** http://www.chessbase.de/Nachrichten/Current/juli2000/dortmund_1907.htm

[CSS00] **Bums - da krachte es.** Computer-Schach & Spiele, Heft 4, (2000).

[CW99] **Schwerpunkt Unternehmenssteuerung.** Computerwoche, Nr. 25, 25. Juni (1999).

[HS00] **Schachprogramm schlägt Großmeister.** http://www.heise.de/newsticker/data/rh-20.07.00-000/

[ICC00] **P.ConNers wins the 10^{th} Grandmaster Tournament in Lippstadt**. ICGA Journal, Vol. 23, No. 3., September (2000)

[IX00] **Schachprogramm schlägt Großmeister.** IX, Heft 9/(2000).

[LS00] **Computer gewinnt Turnier.** Lippstadt am Sonntag, 23. Juli (2000).

[NW00] **Uni-Programm schreibt Schachgeschichte.** Neue Westfälische, 19. Juli (2000).

[PAT00] **Sensations-Coup perfekt.** Der Patriot, 17. Juli (2000).

[S00] *K. Müller.* **Computer gewinnt Kategorie XI-Turnier.** Schach, 8/(2000)

[SCHM00] *A. Brenke, S. Wehmeier.* **Der Anfang vom Ende?** Schach Magazin 64, 15/(2000).

[WT00] **Computer siegt.** Wochentip, 19. Juli (2000).